SAM Driver & Software Architecture overview

Nils Bore
Robotics, Perception and Learning Lab, KTH, Stockholm
nbore@kth.se

***Abstract*—This is a design document intended to guide the development of the software architecture for the SMARC SAM AUV. We detail the interface of the hardware components to the software stack, and propose an interface between the basic software components. The design document is accompanied by implementations of several of the proposed components, as found in the repository `gitr.sys.kth.se/smarc-project/sam_drivers`.**

I. Architecture Overview

The SAM AUV platform is designed to use two identical ARM processors as its main computers. These will be referred to as the *base* and *science* computers respectively. In this document we are mainly concerned with the software architecture on the *base* computer, and any discussion refers to that unless otherwise mentioned. The base computer communicates with sensors and actuators (henceforth referred to as *devices*) via the CAN bus protocol. Attached to most devices are *Teensy* microcontrollers, which handle the interfacing with the CAN bus network. The microcontrollers are also responsible for synching the time of all devices to that of the computers.

II. ROS Middleware

The Robot Operating System (ROS)[1] is a so called *middleware*, that allows different processes on a computer or network to communicate with each other in different ways. It also provides ways of organizing and running software. ROS allows the transmission of data types, called ROS *messages*, between components via a publisher/subscriber model. A large collection of message types are already provided through the ROS software distribution, but you may also define new message types. The advantage of using provided or standardized message types is that the ROS community already provides tools for operating on many of these types, and that you can display them in the *Rviz* visualization tool.

III. UAVCAN Protocol

UAVCAN[2] is a high-level protocol defined on top of CAN bus. It allows for easy and robust communication of data types over a CAN bus network. Similar to ROS, UAVCAN defines its own message types, and also allows the user to extend the system with user-defined message types. UAVCAN *nodes* are responsible for transmitting and receiving messages over the network. They may run both on full-fledged processors such as that of the base computer, or on microcontrollers. UAVCAN also provides tools for synching the time of the attached nodes.

[1]`https://ros.org/`
[2]`https://uavcan.org/`

IV. Components

A. Devices & Teensys

As stated the Section I, most sensors and actuators are hooked up to a Teensy microcontroller. The microcontrollers all provide a UAVCAN node that may provide both subscribe and publish functionality, depending on the capabilities of the attached device. Note that ROS and UAVCAN messages, while similar, can not be defined in a way as to be shared between the two. To get data from the Teensys into the ROS middleware on the base computer, we therefore need to define an interoperability layer.

B. uavcan_ros_bridge

The interoperability layer between ROS and UAVCAN is called *uavcan_ros_bridge*[3]. It is divided into two parts, one that converts UAVCAN messages into ROS messages and one that converts ROS messages into UAVCAN. This division is motivated by the way the ROS and UAVCAN event loops are implemented. Specifically, in the case of the UAVCAN to ROS bridge, the event loop is driven by UAVCAN, and a function to publish a ROS message is triggered every time a UAVCAN message is received. The other way is driven by the ROS event loop in a similar manner. The division minimizes delay in the propagation of messages.

In effect we would like a direct translation between the message types of UAVCAN and ROS. However, the specification of the messages differ on a few points[4]. Luckily, all of these differences can easily be avoided by using the following convention when translating a message in either direction.

- **Variable size int types of UAVCAN**: always use the next largest int size in ROS.
- **ROS topic names**: ROS allows publishing on topics. The topic names should be defined statically in the bridge node.
- **UAVCAN node id recipients**: UAVCAN supports recipients being specified in a message. These should be defined statically in the bridge node.

With these workarounds, any types on either side can always be translated to a suitable type and message on the other side via the bridge.

To convert a UAVCAN message to ROS you need to write a conversion function. The procedure is the same as for conversions in the other direction. The conversion function for the IMU messages might look as follows.

```
template <>
bool convert(
    const uavcan::equipment::ahrs::RawIMU& uav_msg,
    sensor_msgs::Imu& ros_msg)
{
    ros_msg.header.stamp =
        convert_timestamp(uav_msg.timestamp);
    ros_msg.linear_acceleration.x =
        uav_msg.accelerometer_latest[0];
    ros_msg.linear_acceleration.y =
        uav_msg.accelerometer_latest[1];
    ros_msg.linear_acceleration.z =
        uav_msg.accelerometer_latest[2];
    ros_msg.angular_velocity.x =
```

[3]`gitr.sys.kth.se/smarc-project/sam_drivers/tree/master/uavcan_ros_bridge`
[4]`github.com/tum-phoenix/drive_ros_uavcan/README.md`

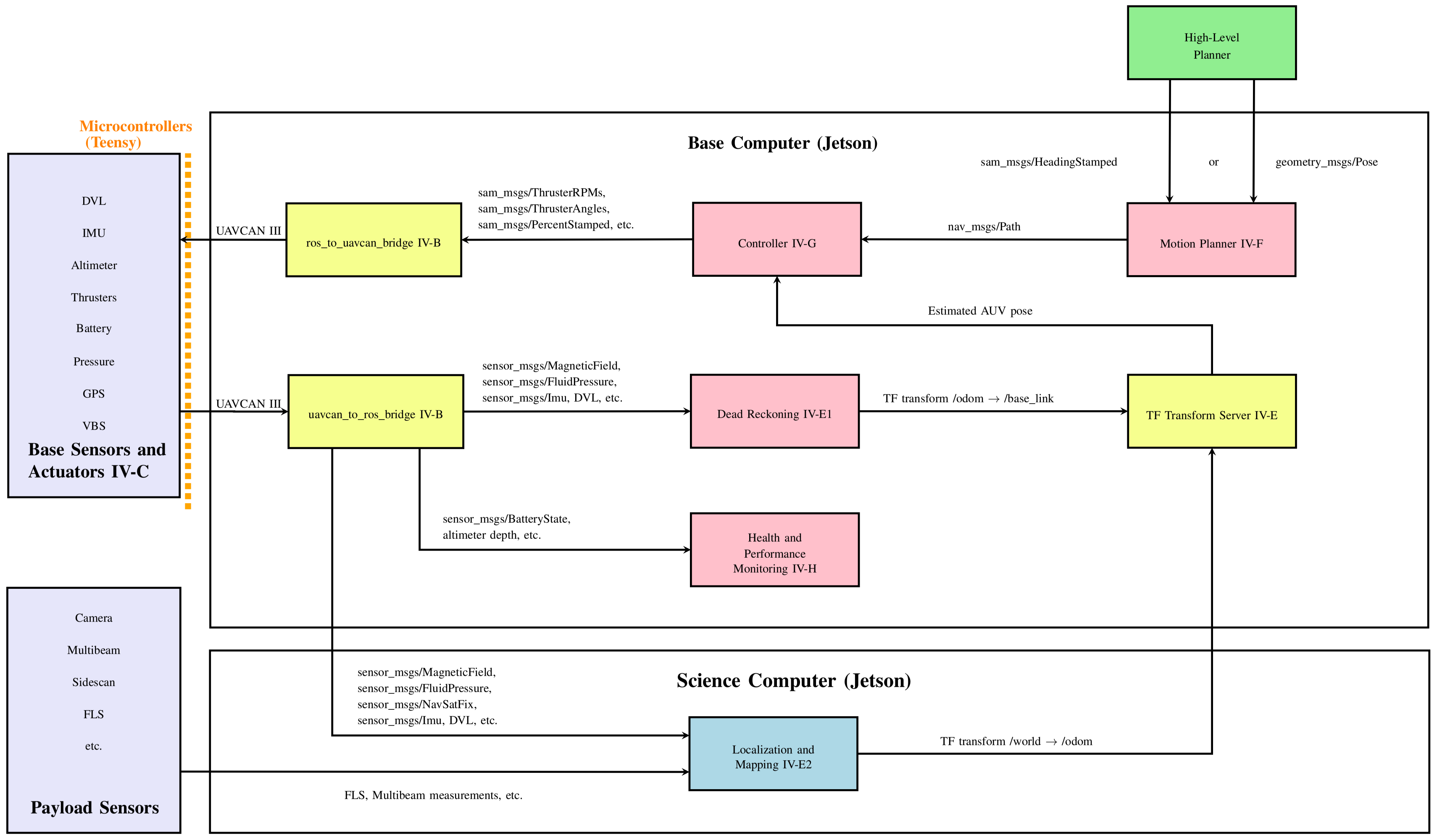
High-Level Planner
Base Computer (Jetson)
sam_msgs/HeadingStamped
or
geometry_msgs/Pose
Microcontrollers (Teensy)
DVL
IMU
Altimeter
Thrusters
Battery
Pressure
GPS
VBS
Base Sensors and Actuators IV-C
UAVCAN III
ros_to_uavcan_bridge IV-B
sam_msgs/ThrusterRPMs, sam_msgs/ThrusterAngles, sam_msgs/PercentStamped, etc.
Controller IV-G
nav_msgs/Path
Motion Planner IV-F
Estimated AUV pose
UAVCAN III
uavcan_to_ros_bridge IV-B
sensor_msgs/MagneticField, sensor_msgs/FluidPressure, sensor_msgs/Imu, DVL, etc.
Dead Reckoning IV-E1
TF transform /odom → /base_link
TF Transform Server IV-E
sensor_msgs/BatteryState, altimeter depth, etc.
Health and Performance Monitoring IV-H
Camera
Multibeam
Sidescan
FLS
etc.
Payload Sensors
Science Computer (Jetson)
sensor_msgs/MagneticField, sensor_msgs/FluidPressure, sensor_msgs/NavSatFix, sensor_msgs/Imu, DVL, etc.
Localization and Mapping IV-E2
TF transform /world → /odom
FLS, Multibeam measurements, etc.

Fig. 1: Overview of the base SAM AUV dataflow, with message types detailed. The Section of the components are indicated next to the labels. Note the orange line next to the base sensors and actuators. It indicates that each component is connected to the CAN network via a Teensy microcontroller, see Section IV-B.

```
        uav_msg.rate_gyro_latest[0];
    ros_msg.angular_velocity.y =
        uav_msg.rate_gyro_latest[1];
    ros_msg.angular_velocity.z =
        uav_msg.rate_gyro_latest[2];
    return true;
}
```

It converts the timestamp, as provided by the Teensy, and sets the values of all the ROS members to the corresponding UAVCAN values. In addition, we need to add the following line in `uavcan_to_ros_bridge.cpp`.

```
uav_to_ros::ConversionServer<
    uavcan::equipment::ahrs::RawIMU,
    sensor_msgs::Imu>
    server(uav_node, ros_node, "uavcan_imu");
```

The string `"uavcan_imu"` tells the program where to publish the produced ROS topic. With this, the `ConversionServer` interface will take care of subscribing to the correct UAVCAN topics and publish a message to ROS every time a new message is received.

C. ROS Sensor Interfaces

Whenever possible, we should use the standard ROS messages to represent the data flowing between components. For simple sensors, there are for the most part suitable message types available. In the case of more complex sensors, and in particular the sonar sensors in the SAM AUV, we will have to implement our own data types. The exact interface of these types will be decided when the sensors are available, and we have drivers to inspect the presented data. For the base sensors, we propose using the following messages.

1) IMU:

type	`sensor_msgs/Imu`
topic	`/imu`

2) Magnetometer:

type	`sensor_msgs/MagneticField`
topic	`/magnetic_field`

3) GPS Fix:

type	`sensor_msgs/NavSatFix`
topic	`/gps_fix`

4) Pressure:

type	`sensor_msgs/FluidPressure`
topic	`/pressure`

D. ROS Actuator Interfaces

Similarly, the actuators of an AUV are specific to the hardware. But since we already know the interface of the actuators, we have prototyped message definitions in the `sam_msgs` repository[5]. We propose the following data types.

1) Thruster RPM:

type `sam_msgs/ThrusterRPMs`
topic `/thruster_rpms`
contents
```
int32 thruster_1_rpm
int32 thruster_2_rpm
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
```

2) Thruster Angles:

type `sam_msgs/ThrusterAngles`
topic `/thruster_angles`
contents
```
float32 thruster_vertical_radians
float32 thruster_horizontal_radians
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
```

3) Ballast Angles:

type `sam_msgs/BallastAngles`
topic `/thruster_angles`
contents
```
float32 weight_1_offset_radians
float32 weight_2_offset_radians
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
```

4) Longitudinal weight position:

type `sam_msgs/PercentStamped`
topic `/longitudinal_weight_position`
contents
```
float32 percent
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
```

5) Buoyancy piston position:

type	`sam_msgs/PercentStamped`
topic	`/buoyancy_piston_position`

E. ROS TF Interface

ROS uses the TF library [6] to represent the geometric state of the world, i.e. the transforms between different parts. In our case, we will likely have at least three notable transform frames, `/world` - the global reference frame, `/base_link` - the position of the AUV center of rotation, `/odom` - the dead reckoning reference frame. The dead reckoning and SLAM systems produce different transforms within this structure. DR produces high-frequency estimates of the `/base_link` pose in the `/odom` reference frame. Meanwhile, SLAM produces lower frequency corrections to this transform by publishing the transform between `/world` and `/odom`. The TF transform server takes care of synchronizing these transforms, and provides us with full transforms between `/world` and `/base_link` at the frequency of DR. This also means that we can run with purely based on DR if there is no SLAM correction available.

1) Dead Reckoning:

type	`TF transformation`
topic	`/tf`
frames	`/odom -> /base_link`

2) SLAM:

type	`TF transformation`
topic	`/tf`
frames	`/world -> /odom`

F. Motion Planner Interface

The motion planner discussed here is the one used for traversing distances under water. There may also be

[5]`https://gitr.sys.kth.se/smarc-project/sam_drivers/tree/master/sam_msgs`

[6]`http://wiki.ros.org/tf`

other planners and controllers e.g. for hovering or for ascent/descent. We mainly have two ways of commanding the AUV to move. First, we may give it a pose in the global reference frame, i.e. `/world`. We may also give it a heading that it will follow. In both cases, we need to supply a target depth of the AUV that it should try to stay at. In both cases, we will use the `z` component of the messages for this purpose. We propose that if `z` has a negative value, it refers to the target depth below the sea floor. In instead `z` is positive, it should be interpreted as the target altitude (height above seafloor). In addition, there are safe altitudes and depths to take into account when planning, see below.

1) Parameters: The motion planner interface uses both ROS topics and ROS parameters. Parameters can be set by any ROS node and also accessed by any node. In particular, the motion planner should take into account the following parameters.

- **Safe altitude** the motion planner should always try to stay above this height over the sea floor

type	`double`
name	`/safe_altitude`

- **Safe depth** the motion planner should always try to stay above this water depth

type	`double`
name	`/safe_depth`

2) Topics: The proposed topic to specify a trajectory are defined as follows. It can either be a heading,

```
type     sam_msgs/HeadingStamped
topic    /nav_heading
contents float64 heading
         # depth if negative,
         # altitude if positive
         float64 z
         std_msgs/Header header
           uint32 seq
           time stamp
           string frame_id
```

or a pose in the `/world` frame,

```
type     geometry_msgs/PoseStamped
topic    /nav_goal
contents std_msgs/Header header
           uint32 seq
           time stamp
           string frame_id
         geometry_msgs/Pose pose
           geometry_msgs/Point position
             float64 x
             float64 y
             # depth if negative,
             # altitude if positive
             float64 z
           geometry_msgs/Quaternion orientation
             float64 x
             float64 y
             float64 z
             float64 w
```

The motion planner then produces a message of type `nav_msgs/Path` (see the next section). The path represents a realistic motion trajectory of the AUV, that can be followed by a controller algorithm. The rationale for using a `nav_msgs/Path` for both kinds of goals (way point and heading) is two-fold:

1) It minimizes development overhead
2) We still need a 3D path in the z-axis, to keep the desired depth or altitude

We anticipate that this will work well in our scenario, since the planner is expected to run every few seconds.

G. Controller Interface

The controller subscribes to the path from the previous section, together with the transform between `/world` and `/base_link`, as provided by the TF server. It then tries to control the actuators of the vehicle to follow the desired path. Note that the controller may be separated into several different controllers, for example for the xy-plane and for the z-axis (depth control). The path is simply a collection of successive poses, as defined below.

```
type     nav_msgs/Path
topic    /global_plan
contents std_msgs/Header header
           uint32 seq
           time stamp
           string frame_id
         geometry_msgs/PoseStamped[] poses
           std_msgs/Header header
             uint32 seq
             time stamp
             string frame_id
           geometry_msgs/Pose pose
             geometry_msgs/Point position
               float64 x
               float64 y
               float64 z
             geometry_msgs/Quaternion orientation
               float64 x
               float64 y
               float64 z
               float64 w
```

H. Performance Monitoring

We briefly touch on performance monitoring here, together with some relevant message types. Similar to the safe depths of Section IV-F, the AUV will also have critical depths/altitude parameters. If these parameters are exceeded, it may choose to do an emergency ascent, or take other measures for safety.

- **Critical altitude** the monitor triggers an emergency ascent when below this altitude

type	`double`
name	`/critical_altitude`

- **Critical depth** the monitor triggers an emergency ascent when below this water depth

type	`double`
name	`/critical_depth`

In addition, by inspecting the health and charge of the battery, it may also choose to abort the mission.

```
type     sensor_msgs/BatteryState
topic    /battery_state
contents std_msgs/Header header
           uint32 seq
           time stamp
           string frame_id
         float32 voltage
         float32 current
         float32 charge
         float32 capacity
         float32 design_capacity
         float32 percentage
         uint8 power_supply_status
         uint8 power_supply_health
         uint8 power_supply_technology
```

```
bool present
float32[] cell_voltage
string location
string serial_number
```

V. Summary

We have presented a detailed overview of the components needed for the SAM base navigation system. It may now be of interest to look at the reference implementations in `gitr.sys.kth.se/smarc-project/sam_drivers`.