RtBot: a signal processing framework focused on early event detection

Robert Carcassés Quevedo, Eduardo Pérez Verdecia, Yuriel Núñez Jiménez

Abstract

RtBot framework is presented as a general purpose digital signal processing framework focused on event detection.

Keywords: digital signal processing, open source, event detection

1. Introduction

RtBot is a general purpose digital signal processing framework focused on early event detection. It's written in c++ and has been built with performance in mind.

Wrapper libraries exist for python, javascript (using wasm) and rust, and in the future we plan to support java and swift languages as well. In this sense RtBot can be easily adapted in any existing codebase, as it already provides interfaces to the most popular existing computer languages or new ones can be built for those missing languages using their native interface with c++. RtBot programs are very fast and have a low footprint, being able to run even in microcontrollers with very limited resources, such a raspberry pi pico or the esp32.

As a digital processing framework, it contains the common pieces found in such frameworks such as high and low pass filters, finite response filters, signal re-samplers and so on.

As an event detector framework it is designed to give developers the tools to minimize the time difference between the moment when the event is detected and the moment when the event actually happened.

2. Architecture

2.1. Modules

RtBot framework is divided into a set of modules: *core*, *api* and *std* ones. This list is likely to grow in the future and users can also extend it with their own private business model specific ones.

The *core* module contains the general definitions and it does not depend on any other. You can write a program using only the core module, but you will have to write it in c++ and explicitly construct the operators and their connections.

The *api* module allows users to easily interact with the core module. For instance it can read RtBot programs saved in json format and construct the correspondent internal pipeline of operators at runtime. In this module is also where the external bindings are defined, giving a simplified api to run programs. If you are developing a web or desktop app, or even a mobile app, you will likely use this module. On the other hand, if you are building an embedded application

for a microcontroller, including this module in your build will cause a large binary output that probably won't fit on the device. In this case you will have to rely only on the core and std modules to make sure you produce binaries small enough.

The *std* module represents a standard library of commonly used operators that can be used to build programs. Operators defined here are general purpose ones, like the inputs, resampler or finite response operators, which are ubiquitous used in any kinds of input signals. In practice this module and the core one will be the minimal components you will need in order to create some non trivial program.

2.2. Inside the core

An RtBot program expects a *signal* as input, which is a time-ordered stream of messages to be fed to it, one at the time, in what we call a *program iteration*. This iteration may or may not produce an output, which will be defined later.

A *message* is a tuple of timestamp and value. It is important to remark, even if obvious, that the timestamp in the message is the one used for any internal analysis, and not the time that the message arrives. RtBot has no clock inside or anything similar, so it has no way of counting real life time.

RtBot assumes the time inside a message to be an integer. The main motivation for this is that comparing two timestamps is more efficient and meaningful in this case than if we allow the timestamps to be of float type. This allows, for instance, to synchronize signals according to its timestamp in a much simplified and efficient way. This shall not, however, be a hurdle to adopt RtBot to process signals with fractionary timestamps, as a rescaling in the time dimension of the signal prior to sending it to the RtBot program will solve any type mismatch. This time scale, i.e. seconds or milliseconds, depends on the nature of the signal being processed and it is up to the programmer to decide which makes sense for the given type of signal.

Vectors are represented as a *burst of messages* that all have the same timestamp, where the emission order matches the vector components' order, being the last received correspondent with the last index.

2.3. Operators

The main concept in the RtBot architecture is that of an operator. An operator it's simply a computational unit that takes an input, transforms it and produces an output. Operators take messages as inputs and produce other messages as outputs. Remarkably, operators assume that they will receive messages with increasing timestamps, this is, ordered in time.

Operators can be connected with other operators, in a directed way, such that one operator can have children operators. If an operator has children then whenever it produces an output it will be sent to its children which will consider those messages as its own input, and so on. Operators may change the throughput of the stream they receive, producing any number of messages, or none, per each message it receives. In this sense they are also flow controllers, deciding when and what to forward to its children operators.

2.3.1. Input

An RtBot program is then this set of operators connected in a graph structure, a pipeline, where the external signal entry points are marked by the input operators. An input operator is just an operator that is designed to be a good intake of the external signal. Input operators are not mandatory but recommended entry points, as they perform sanity checks on the input

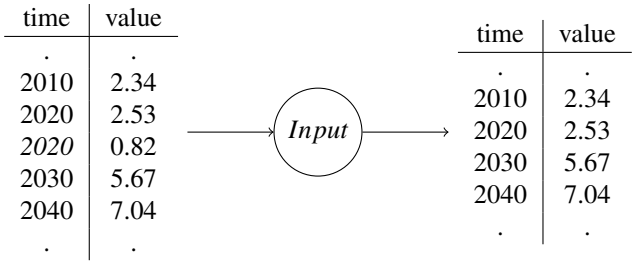

Figure 1: Action of the *Input* operator in the incoming signal. Notice how the second message with the same timestamp $t = 2020$ is ignored. The same would happen if the message timestamp would have been less than the previous one.

signal before forwarding the message to others. For example, an input operator will discard new messages with timestamps prior or equal to the last message received, guaranteeing the time order in the messages it forwards.

2.3.2. Resamplers

All operators that implement some math over their input assume that the messages it will receive are not only time ordered but also *equidistant in time*, or *equally spaced in time*. We will call this type of signal a regular one, while those that don't fulfill this will be named irregular. This is a strong assumption and it will likely be the case that input signals are irregular. This is why we have a special set of operators called resamplers.

A resampler operator will take an irregular input signal and will produce a regular one. Internally, it will use the irregular signal as its source of truth, to find a realistic interpolation over the time grid points. The interpolation algorithm depends on the resampler used. In practice this means that, while not strictly necessary, programs will start with input operators followed by resampler ones.

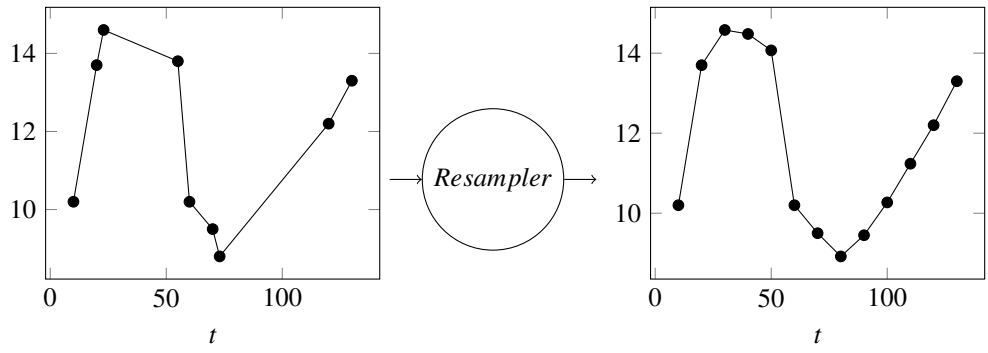

Figure 2: Action of a resampler operator. On the left an irregular signal, whose messages are not uniformly distributed in time. On the right the output of the resampler operator: the produced stream of messages now are equidistant in time.

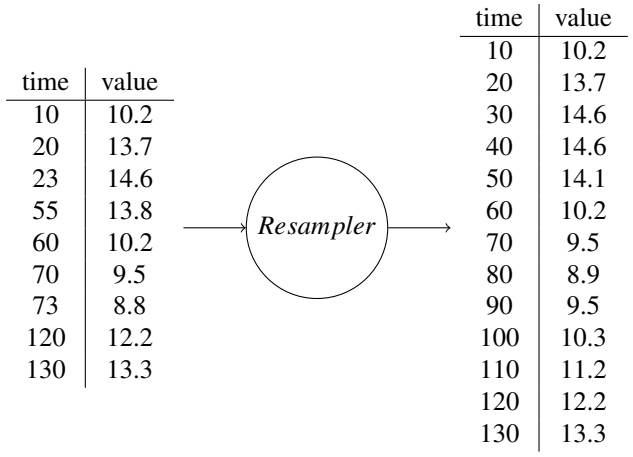

Figure 3: Action of the *Resampler* operator in the incoming signal. This is the same data presented in figure 2.3.2 but in tabular format. Notice how the signal gets augmented by the resampler in order to fulfill the requirement of emitting regular time intervals. In this case the grid step is $\Delta t = 10$.

2.3.3. Join operators

Join operators combine two streams of messages into a single one. It tries to resemble sql join operation, where two tables are combined into a new one. They are perhaps RtBot special ingredients, and will be used whenever we need some sort of synchronization to happen between two signals.

These operators can have multiple inputs, each one bound internally to a port. They can have any number of *ports*, or input channels, from where the streams of other operators can be received. Messages received through the ports are queued efficiently inside the *Join* operator. In the default mode, the operator will emit whenever it receives a message in any of the ports with time t and there exists *at least one message* across *all* the queues with that time. In such a scenario, all these messages with same timestamp are popped out from the internal queue, and emitted in a burst of messages. The queue also remove all the old messages that there is no chance that can be emitted because they are too old, keeping the process memory efficient. A visual example can be seen in figure 2.3.3.

3. Use cases

3.1. Network traffic monitoring and statistics

3.2. Real time biofeedback

In recent years we have experienced an explosion regarding wearable devices with the capacity to constantly monitor our health. They can measure our heart rate, oxygen or sugar level in blood, for instance. They constantly gather information about our bodies and can offer us feedback about how we performed in certain activities, like how well did we sleep last night and can provide suggestions for our day accordingly.

Cheap wristbands, through its *photoplethysmography* sensor, commonly known as PPG, are able to detect heart pulses by injecting light into our tissue and measuring the luminosity changes

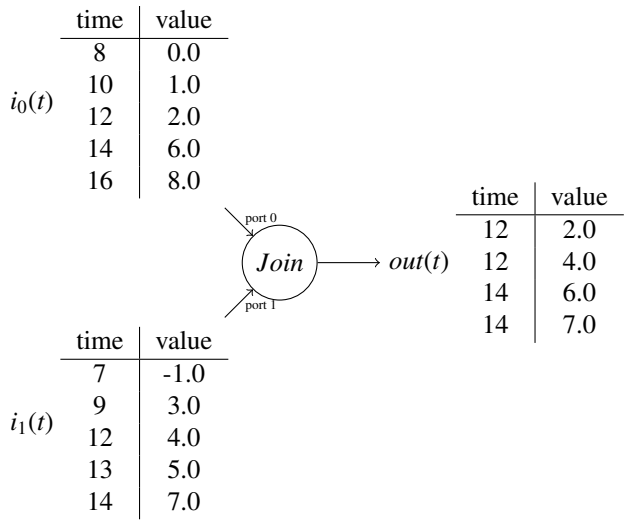

Figure 4: Action of the *Join* operator in the incoming signals. *Join* takes the two signals and emit values with the same time in both input signals $i_0(t)$ and $i_1(t)$, like $t = 12$ in the present example. The output will be a sequence of messages with the same t, ordered by incoming port number, and the value equal to the incoming signal value for the matched time.

of the reflected light. As the blood goes through our veins, it causes a measurable variation in this luminosity, phenomena that can be used to infer our heart rate. From the patterns of variability of the heart rate in time, and also combining this with data of other sensors, many other things can be derived. The device can know whether we are running, sleeping or exercising, or even count how many repetitions of a given exercise type we have done in the gym.

There is one big caveat though: the measurements coming out of these devices often do not reflect reality and it is full of artifacts. There are many causes to this, like wearable devices not being used properly, the quality of the sensor or having a low battery. But even the underlying physics itself of the measuring mechanism is challenging if we want to get a reliable measurement. For example, the shape of the signal coming out of a PPG sensor, the one that doctors put you on the tip of your finger when you first arrive at urgent care, is considerably different, and less trustworthy, than the signal coming out of an *electrocardiogram* (ECG) equipment, which finds your heart rate by measuring the electrical signals that propagate over your body when the heart beats. The last one has sharper peaks compared with the first one, which are easier to detect, leading to better results. It comes to no surprise then that ECG equipment is the standard found in intensive care units.

In any case measurement's quality can be improved in many ways, like for instance improving the wearable analytics software capabilities. RtBot has been designed specifically to excel in this type of environment where cpu and memory are scarce. Its low resource usage footprint, together with the ecosystem productivity tools, allows it to easily create optimized algorithms to get best results per device type. It can help to improve peak detection as well as to flag signal parts as unusable, preventing the output of nonsense measurements. Moreover, due to the broad scope of the operators provided by the framework, it also opens the door for more complex analytics like secondary peak detection, pulse shape and more which can provide valuable

information about our physical and even mental health.

The same argument applies to a large extent to professional health devices like ECG. RtBot can be used as well on their software to provide better analytics while providing a short development cycle from conceptualization to production phases.

3.3. Trading bots

Trading bots are computer programs that trade on behalf of humans in financial markets. They aim to profit by buying or selling some assets timely. This type of trading works best at time scales where technical analysis rather than fundamental analysis is relevant, usually in the scale of hours and down to the microseconds.

Trading bots perform decisions according to the historical data received in the recent past. Trading strategies usually involve watching some indicator, or set of indicators, and to trigger bull or sell orders according to whether these indicators fulfill certain conditions. This fits well into the pattern of RtBot as a framework for event detection. In this case the events would be for instance an indicator going beyond certain predefined boundaries, indicating that it is a good moment for selling or buying an asset. Programmers can use RtBot building blocks to build their strategy and to get as outputs signals that trigger market orders accordingly. For the broad public perhaps more important is the fact that, as RtBot framework is an open source software, they can retain ownership of their trading ideas by running everything locally. This way it can be guaranteed that there is no one observing in secret the bot code, as it could potentially happen when it's being backtested on a cloud environment.

While not completely developed yet, we plan to release a finance module that will contain most of the more useful trading indicators, in addition to other resources that would allow the execution and monitoring of the bot locally, where broker api secrets can be securely configured, for instance.

References

[1]