**FFDE Network**

Foundation For Data Exchange

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

While planning the architecture of a complex mechatronic system it is needed to consider the way in which separate software and hardware components will exchange data and commands. For various designs different requirements can be defined. This paper discusses a software layer for exchanging data between software modules of a drone.

Jakub Mnich – MikroCpp (Universal Flying Platforms) 25 August 2016

1. **Defining the problem**

A drone is an unmanned flying vehicle capable of executing complex tasks without human intervention. In this paper it is assumed that discussed technical solutions apply for a quadrotor vehicle designed with autonomous operation in mind. Also such a vehicle should be capable of carrying advanced onboard computer, for example ARM-based Linux single board computer. (Raspberry Pi in this particular case)

With the access to an extensive computing power offered by a modern single board computer with multi-core CPU usage of high level tools and software components embedded in OS like Linux can be considered. This gives a great opportunity to implement truly universal solutions functionality of which can be effortlessly expanded in future. However such an approach requires deep consideration as every decision made on early design stage will affect future components and some of this decisions cannot be easily changed without modifying large parts of code or hardware.

The very first step in design process would be to understand technical issues of a copter:

1. Copters are highly unstable vehicles and as such require low-latency communications between their software and hardware modules. Delays in data transmission higher than single milliseconds are therefore unacceptable.
2. Transmitted data needs to reach its destination quickly but the magnitude of information flow is relatively small. Most information comes from simple sensor readings which are further processed and passed between various software modules.
3. Above statements indicate that it is extremely important to enable versatile yet fast access to relatively small amounts of data that is being exchanged between numerous software modules. This can be achieved with the use of modern object-oriented programming tools.
4. **Solution concept**

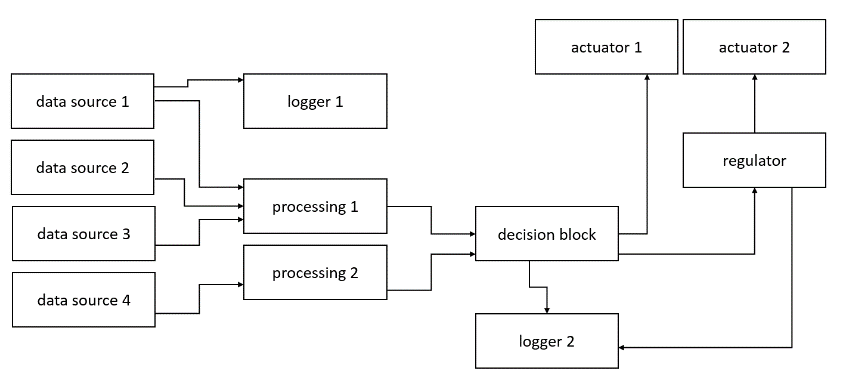
Typical solution for exchanging data between software components in a drone looks similarly to the following diagram:

Fig. 1– typical solution to data exchange issue in existing copters

This approach ensures that the transmission time from module to module is the shortest possible as every functionality is encapsulated in a class and instantiated by initialization procedure. This allows modules to interact in a very direct way by calling methods on members being lower in hierarchy.

Simple as this solution is it does not offer flexibility often desired in experimental machines where adding extra features or modifying existing parts of code happens on daily basis, often without notifying all members of the team.

Desired flexibility can be achieved by rethinking the whole onboard software structure:

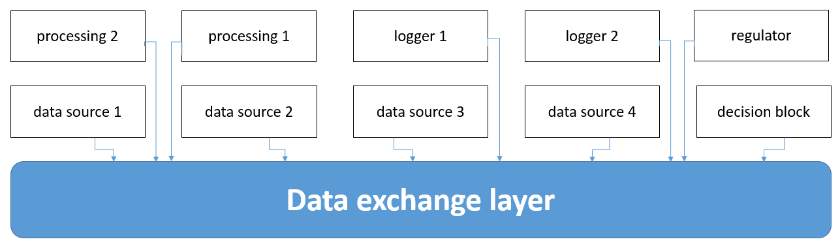
1. Functional blocks should not be represented by different classes in one program but rather be divided into many separate programs running simultaneously on the onboard computer. This could allow extremely easy modification of the code as only one program would require recompilation after making changes as long as the provided API remains compatible.
2. Dividing software into several programs solves flexibility issue but it also creates a new one - separate blocks cannot communicate directly. This problem can be solved by introducing an explicit data exchange layer.

Fig. 2 - system reorganized with a data exchange layer

Such a layer would provide a common API to every program that shares the layer therefore allowing them to easily exchange information without direct interaction. This particular feature means that the only API all programs need to implement is the API of data exchange layer. It allows a whole functional block to be either modified or completely replaced, as long as it can deal with incoming data, without any modifications of other modules required.

There are a few ways to provide an access to the API to the client programs (nodes). The easiest to implement is to provide an object to be instantiated by every node and used as a terminal connecting it to the layer. In order to keep such a solution maintenance-free it is also required to introduce a central node, connection to which would be available to every node with no additional information provided to the terminal object after startup.

From now on the following naming convention will be used: client program of the layer – node, terminal object – FFDEServer or server, central node – kernel and the whole data exchange layer – FFDE (Foundation For Data Exchange) Network.

The internal structure’s flexibility of the layer can be achieved in a few different ways: common configuration file, fixed addressing kept by each node or by dynamic declarations of all nodes. For FFDE the last option is an obvious choice as it allows to keep it maintenance-free. The whole internal structure of the network can be built only with the information provided by the nodes to their servers by calling special methods on them. For example, a simple method: connect(node) can be imagined, which creates a connection through the layer between the node which called this method on its server and the node which name was passed in the argument.

The data exchange itself consists of two different activities: transmission and reception. Due to their nature they require completely different handling. For FFDE Network the following approach has been selected:

* Transmission is implemented with a set of methods provided to the node by its server.
* Reception activities are encapsulated in an event object which is then handled by dedicated FFDE Event system**.**

1. **Tools**

* Means of transferring information

Any data exchange layer requires a lower level mechanism that allows to transfer data around the system. Network sockets have been chosen for this purpose being a solution with satisfying performance, high versatility and rich API delivered with Java JDK.

* Programming language

The right choice of a programming language has a great impact on development time and serviceability of the created code.

Three languages were considered for the purpose of this project: C++, Python and Java. All three of them are fully supported by Raspberry Pi.

C++ offers a great performance of applications but it results in increased development time and problems with debugging complex software. Moreover the code, once completed, is relatively difficult to modify.

Fig. 3 - data flow in the subscription system

Python allows to write programs quickly and easily modify them in the future. Its main drawbacks are low performance and GIL.

Java delivers both high development speed and good code serviceability. The performance is more than satisfying. For these reasons Java 8 has been chosen for the purpose of writing FFDE Network.

* Integrated development environment

There are more IDEs for Java available on the market than can possibly be mentioned in this paper so author decided to use the one that matches best his personal preferences. The selected IDE is IntelliJ from Jet Brains.

IntelliJ provides simple user interface and more intelligent auto-completion than Eclipse or NetBeans which speeds up the development process.

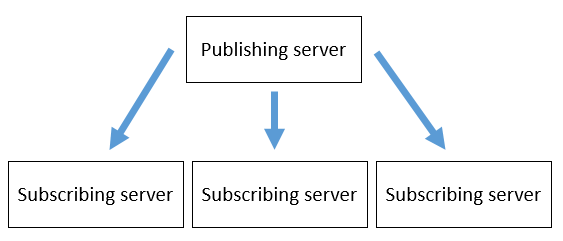
* Additional troubleshooting tools

Standard debugger built in IDE is a great tool but it is often good to have additional application capable of tracking the program activity in real time. For this purpose VisualVM has been selected.

1. **Implementation**

FFDE Network implementation **[1]** consists of nearly 2000 lines of Java code and Javadoc comments. All classes required to use it are located inside FFDENetwork package.

FFDE Network implements a few different models of exchanging data between clients. All are described in the following section:

1. Subscription system

Subscriptions allow to easily implement a module producing data on regular basis which does not require feedback from its recipients. The examples of such modules are simple sensor or data filter.

In order to create a subscription the node informs its server that it publishes a data channel. The server then registers that channel in the kernel and setups a special thread responsible for handling subscribers.

In order to become a subscriber the node calls a method like: subscribe(subscription address) on its server. Then the server queries kernel for the port where requested subscription is available and automatically setups both the connection and the event system for the node.

After establishing connection the publishing node can update its data channels with fresh data causing all subscribers to receive an event with this data.

The important feature of this subscription system is its flexibility. Once a module is implemented with the use of it, adding new recipients of its data will never require modifying the code as the publisher does not require information how many subscribers it’s going to have. This makes activities like logging or extending processing chain fast and easy.

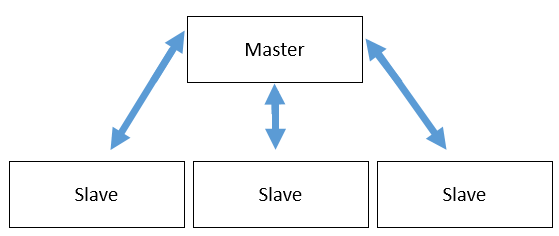
1. Master – slave relation

Fig. 4 - data flow in master - slave relation

This way of exchanging data is designed for special uses when a clear hierarchy between several modules can be defined. For example this would apply to a central unit controlling many algorithms processing the same data – making iterations with different parameters perhaps, where this central unit would be master and separate algorithms its slaves. This approach would allow to divide slave algorithms into separate programs making it easier for a team of developers to cooperate.

FFDE allows for two way data transmission in this relation. Master can control any number of slaves while a slave can have only one master. There are no special functionalities designed for additional interactions between master and slaves. The purpose of this relation is mostly architectural.

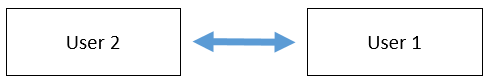
1. Pipelines

Fig. 5 - data flow in a pipeline

Pipeline feature is the last added to FFDE Network and is mainly based on master – slave relation but it provides more freedom to the user as no hierarchy needs to be implemented here.

A pipeline is basically a connection between any two modules registered in the same FFDE Network. The users (nodes) can exchange data just by calling a method on their servers. A pipeline has only two ends so no addressing is needed. Only a name of the other user is needed in order to transmit data to him because a server can be plugged to any number of pipelines.

States of FFDE Network

A situation when a server tries to subscribe a channel that has not been published yet can easily be imagined. There are more conflict patterns that just this one. In order to prevent such situations a set of different states has been introduced. The state of the whole network is controlled by the kernel and servers change it only on its command. Transmitting data through the network is not possible until it reaches its safe state. Four states have been defined:

1. Registration

While the network is in this state all nodes are being started and register themselves in the kernel so that it knows they exist.

1. Declaration

When network reaches this state all servers declare what kind of relations were demanded from them by their nodes. Kernel now assigns ports for different services and registers all opened channels.

1. Subscription

Now that all published channels are known each server can inform kernel which of them it needs to subscribe.

1. Frozen

After the internal structure of the network is established the data transmission can begin. Since this state is reached no changes can occur in the network until restart.

After a node instantiated its server it immediately gives it all information regarding required resources and the server takes care of acquiring them correctly. It also provides a method waitUntilNetworkIsReady() which locks the thread until the FFDE Network is frozen.

Above means that once the network is started there is no way to reorganize this as it would need to get through all these states again. FFDE Network is therefore a static solution with no ability to regenerate after one module fails. The only way to restore it after failure is to reinstantiate all servers and the kernel. This feature is rather undesirable in a drone but after a crush of one of the programs recovering it would be so hard that such emergency procedures would not be implemented in experimental machines for which FFDE is developed.

1. **Usage**

Using FFDE Network is very simple and integrating it into ones code can be covered in a few steps.

1. Provide a kernel

Only one kernel must be present in the FFDE Network. It can run as a separate program or be instantiated together with one of the servers.

The only step needed to create a kernel is to instantiate it with the following constructor:

FFDEKernel(port)

Port is the number of the port where the kernel is going to accept communication from the servers.

1. Instantiate a server

The servers are instantiated with the following constructor:

FFDEServer(name, kernel\_port, node)

Name is the unique name of the node that will be used to register it in the kernel.

Kernel\_port is a number of the port where default connection to the kernel is available.

Node is the reference to the object implementing FFDEObserver interface used to receive events from the server.

1. Declare resources

Every resource can be ordered from the server with a proper method. For example:

ffdeServer.openPipeline(user)

This method opens a pipeline connecting the owner of this server to the user. It works analogically for every other resource.

1. Add handlers for incoming reception events

Incoming reception event is passed to the observer as an argument in the following method from FFDEObserver interface:

public void notifyFFDE(FFDEEvent event)

The correct way to handle an event inside this method is to extract its ID with the following method:

event.getIdentifier()

The identifier is a string that informs which channel delivered it into the server. It allows to identify the source of data. In order to handle the event it is needed to know its identifier.

The following identifiers are used in FFDE Servers:

* “krx” – reserved for communications with the kernel
* “ktx” – reserved for communications with the kernel
* “crx” – reception event from the master
* “srx\_” + <name> - reception event from the slave named <name>
* “pipe\_” + <name> - reception event from the pipeline connected to node <name>
* “sub\_” + <publisher> + “.” + <channel> - reception event from the subscription channel <channel> published by the node <publisher>

For example: subscribing a channel “data” from publishing node named “filter” would produce the following identifier: “sub\_filter.data” .

The recommended way to handle events in a program looks like this:

@Override

public void notifyFFDE(FFDEEvent event) {

switch(event.getIdentifier()) {

case “sub\_x.y”:

handler1(event);

break;

case “pipe\_z”:

handler2(event);

break;

default:

// unexpected event

}

}

FFDE Servers are configured to use only one thread to update the node with reception events in order to prevent synchronization issues but this implies that handlers must not contain time-consuming tasks. Otherwise further updates can be delayed until that task finishes.

1. Wait until the network is ready

The last thing is to ensure that the node will not attempt to use its server until the FFDE Network is ready to handle data exchange. FFDE does not provide any safety mechanism to handle such a situation so avoiding it is up to the user.

FFDE Servers provide however a method which locks the thread until the network is ready:

waitUntilNetworkIsReady()

**References:**

1. Github repository with FFDE Network [https://github.com/jmnich/UFP\_FFDE]