2019/2020

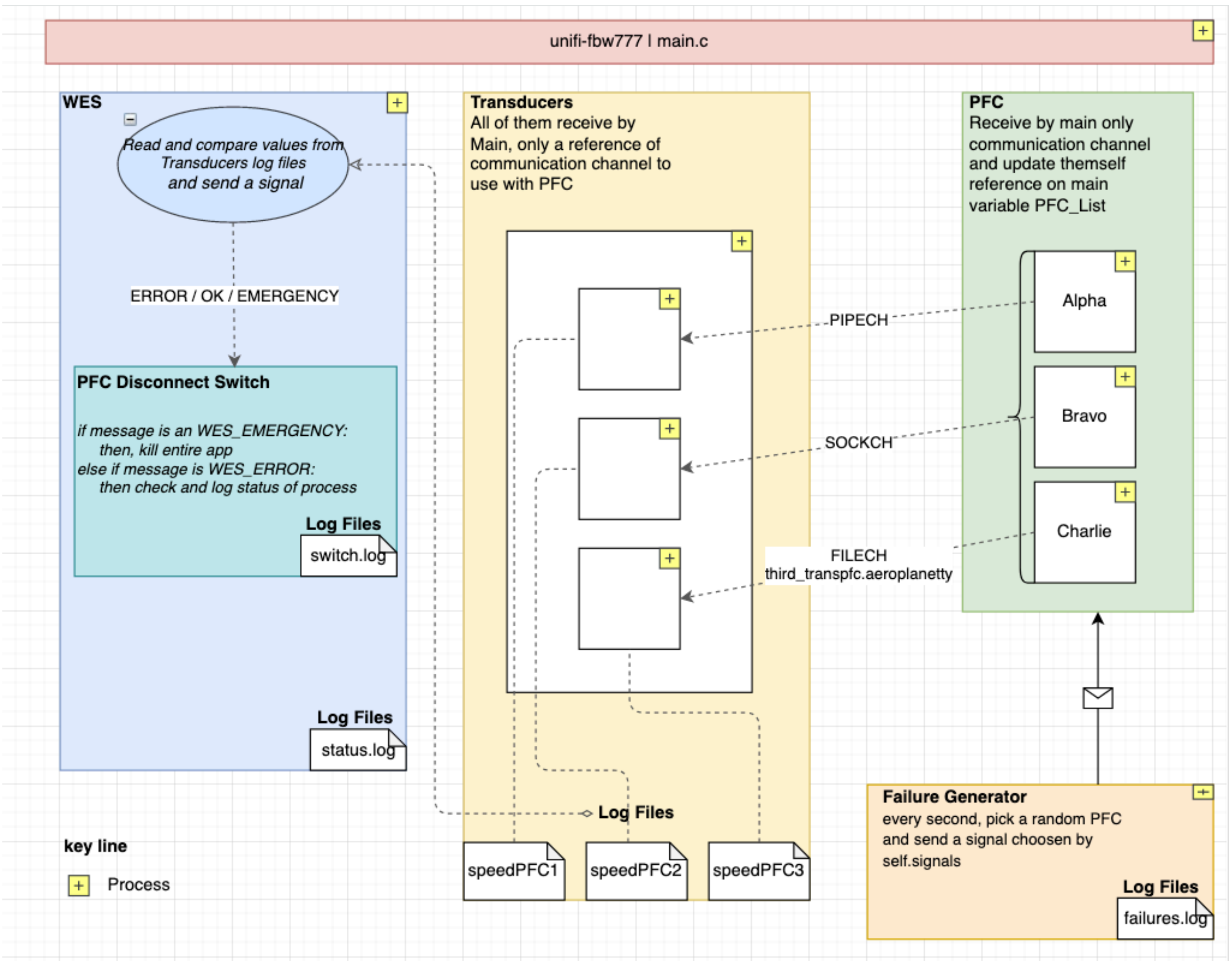
FLYBYWIRE

лететь по проводам

Technical Paper

Graphical explanation of running processes

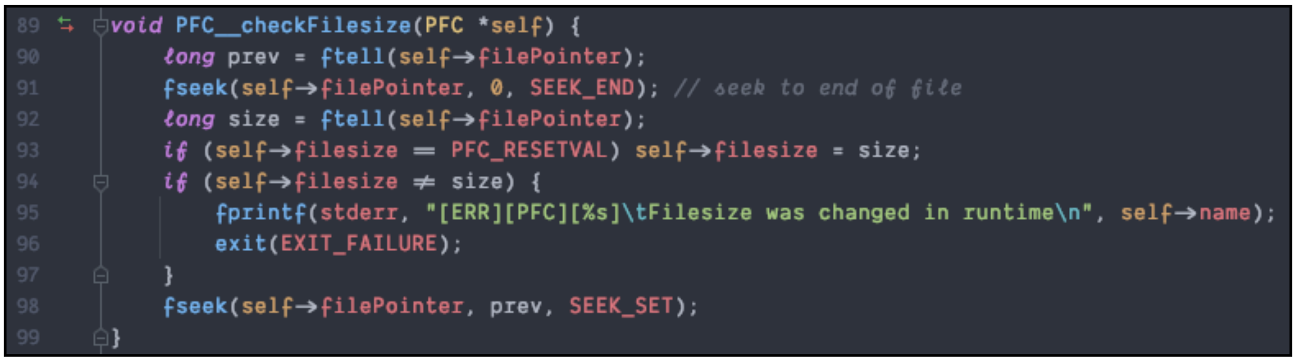
In the following document, each PFC has a unique name, which are respectively Alpha, Bravo and Charlie. In this way we can distinguish which one uses a specific type of communication as will see below.


unifi-fbw777 | main.c
WES
Read and compare values from Transducers log files and send a signal
ERROR / OK / EMERGENCY
PFC Disconnect Switch
if message is an WES_EMERGENCY: then, kill entire app
else if message is WES_ERROR: then check and log status of process
Log Files
switch.log
Log Files
status.log
Transducers
All of them receive by Main, only a reference of communication channel to use with PFC
PIPECH
SOCKCH
FILECH
third_transpfc.aeroplanetty
Log Files
speedPFC1
speedPFC2
speedPFC3
PFC
Receive by main only communication channel and update themself reference on main variable PFC_List
Alpha
Bravo
Charlie
Failure Generator
every second, pick a random PFC and send a signal choosen by self.signals
Log Files
failures.log
key line
Process

Security tips

Filesize check

A "filesize security checker" has been implemented in order to continuously check the size of the files handled by the PFCs. During the execution of the program, each PFC checks the size 3 times per second.
Every PFC after calculation of speed and distance are involved into a new phase, the communication and log. Before sending data to the Transducer, using a specific communication channel (defined in `self→com`), a few parameters are saved into internal attributes, a fileseek of internal file pointer, and an integer representing the file size. However, if at the next read of file, the size is changed, maybe another process is doing an attack to our system. By design no-one process changes file size. ~~Saving also the file seek is useful to a reboot phase of PFC (not implemented)~~.



```c
89 void PFC__checkFilesize(PFC *self) {
90     long prev = ftell(self->filePointer);
91     fseek(self->filePointer, 0, SEEK_END); // seek to end of file
92     long size = ftell(self->filePointer);
93     if (self->filesize == PFC_RESETVAL) self->filesize = size;
94     if (self->filesize != size) {
95         fprintf(stderr, "[ERR][PFC][%s]\tFilesize was changed in runtime\n", self->name);
96         exit(EXIT_FAILURE);
97     }
98     fseek(self->filePointer, prev, SEEK_SET);
99 }
```

See online [line 89]

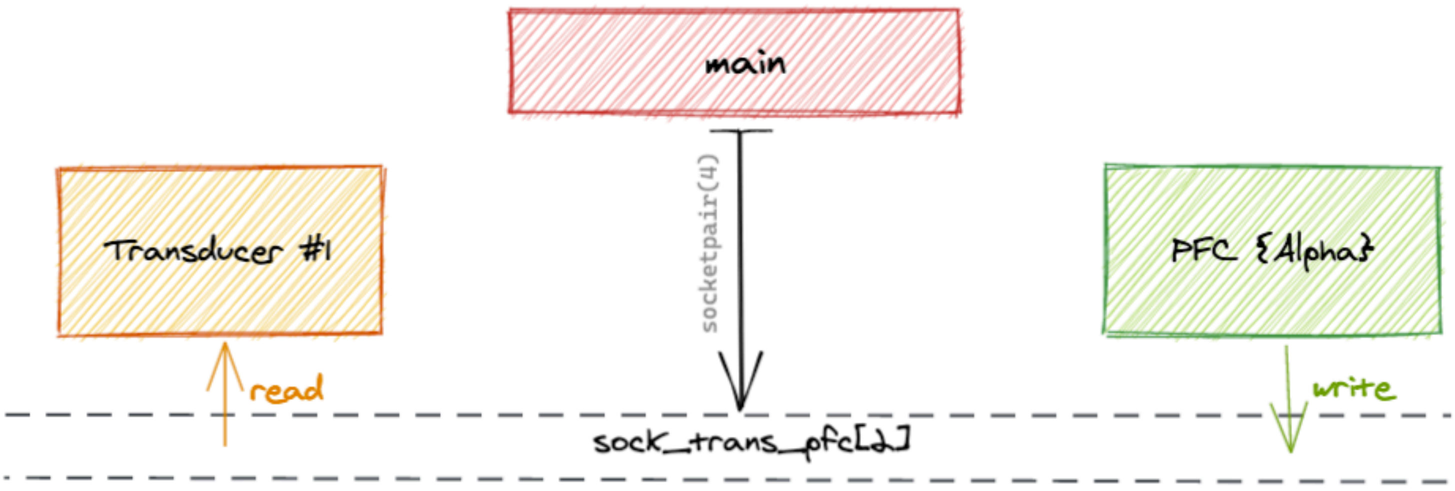
For this check a backup of seek point is stored in variable `prev` then filepointer goes to the end of file and with an `ftell(1)` a file size is determined and stored.

InterProcess Communication techniques, where and why

Socket

By design a "socket based communication" is mandatory between a PFC and Transducer. In this project every communication type is defined in `Channel→channelType`, and in this case, at array of PFC pointers defined in `main.c`, the first one 'PFC_list[0]' has a channel identified by tag SOCKCH (see line 47 in main.c)
As seen in general overview of project. Sockets used by first PFC are created by `socketpair(4)` imported by `socket.h`.
A pair of sockets are created by the process `main` and they are shared with children like PFC and Transducer by using `sock_trans_pfc`.

At PFC process the first action done on that array is `close()` of one side of socket and referred other one into `Channel` of this PFC. At the end also this side `sock_trans_pfc[1]` will closed.



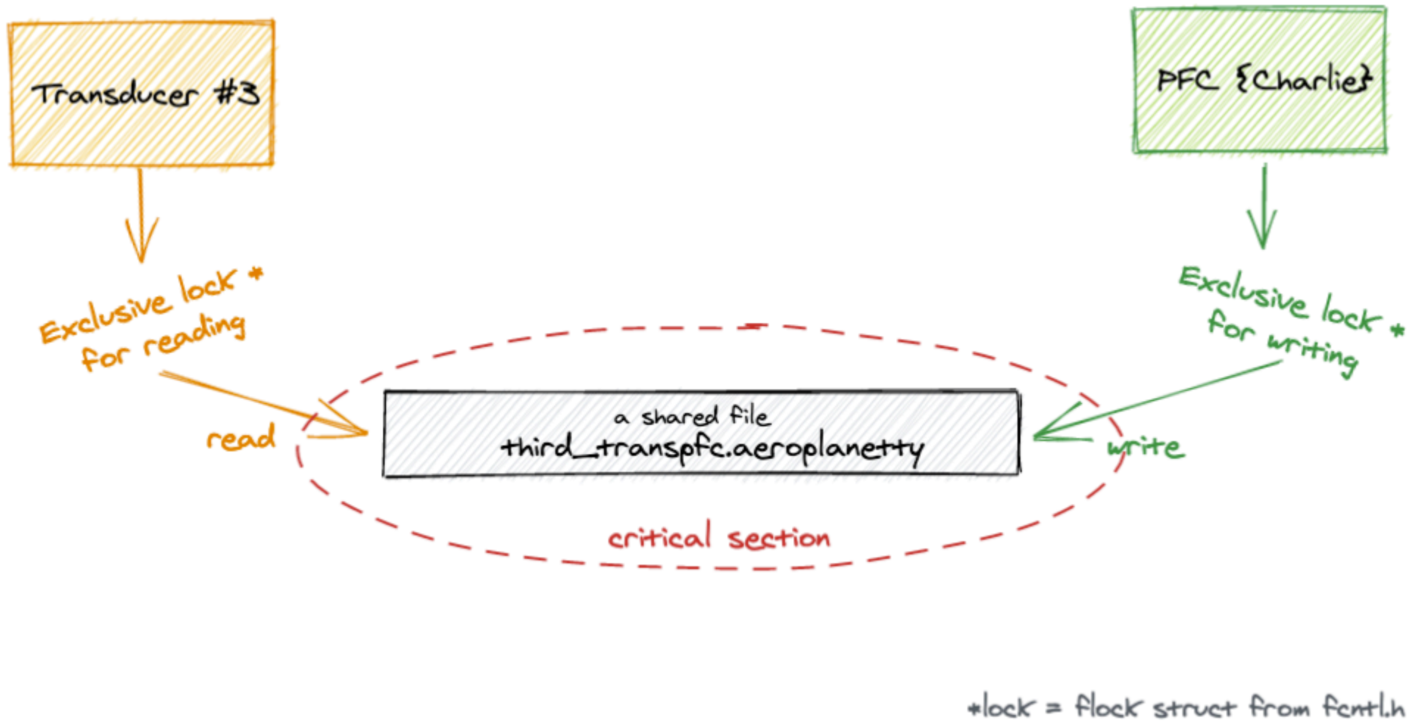
Anonymous Pipe

Same as socket, main process create an unnamed pipe and stores it to variable `Pipe trans_pfc`. In this project all unnamed pipe are managed via Pipe struct defined in utilities.h. This type of communication is very similar to socket, because I/O functions present similar interfaces. The most important difference between this two types of communication is that pipes offer unidirectional communication and sockets offer bidirectional communication. In this project it has been chosen to treat it (sockets) as pipes, to obtain a less complex and more readable code.

Notice: Socketpairs are normal AF_UNIX domain sockets and establishes a connection between them.

Lock on file

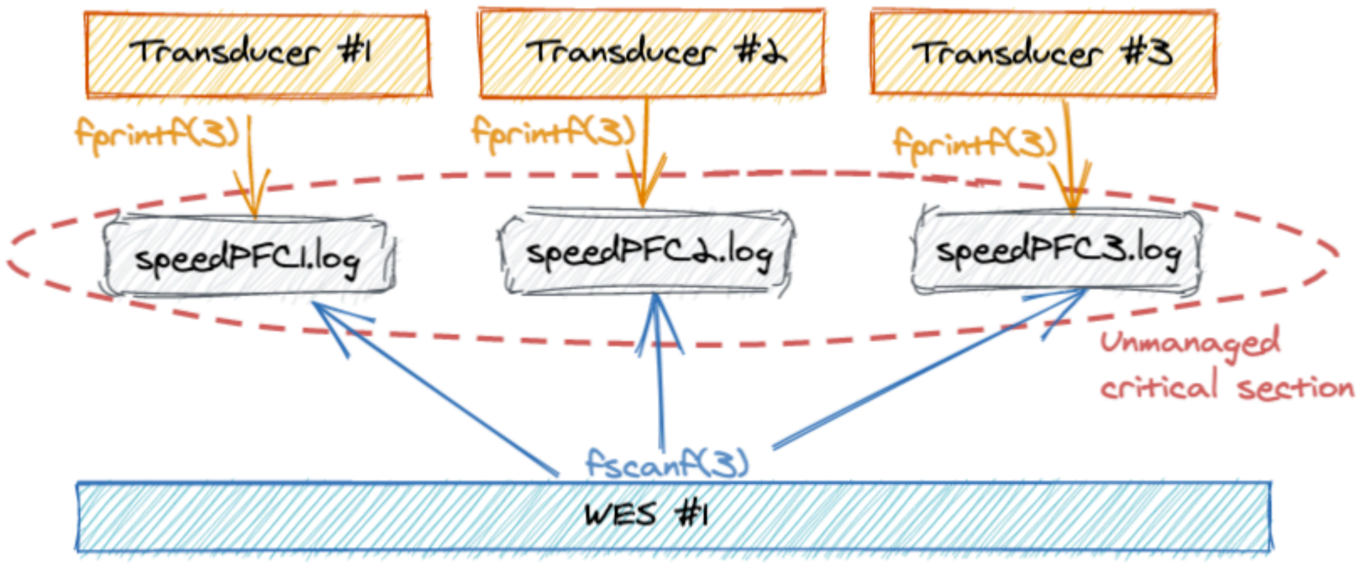
To obtain a fluid communication and a good management of critic section, the communication between one of PFC (Charlie) and one Transducer, is developed with File Locks. By design these two processes need to communicate with a shared file.



Once per second, PFC requires a `F_WRLCK` lock on file, and after writes speed release lock with `F_UNLCK` [block of code line 192], then a Transducer process requires a lock in `F_RDLCK` mode (to read) and after reads speed, release lock at the same way as PFC. This technique allows to solve the potential race condition over the critical section of the shared file.

No-lock on file (time synchronized)

The last technique implemented in this project is available between Transducer and Wes processes. It's been deliberately chosen to not implement any "lock strategies" and let the processes synchornize just using time functions provided by the operating system.



In this case, every Transducer process writes once per second into its logFile, then WES process repeats the execution of `` `fscanf(3)` `` instruction. [see code line 77]

> Notice: *there is the* `_Noreturn` *keyword in the signature of the method Wes__startReading in code above, this mean does not return by executing the return statement or by reaching the end of the function body. Keyword increase readability of code*

With "once per second" is intended a loop instruction with a `sleep(1)`. In this case, there is a bug by design, because WES could miss a line and generate an `` `error` `` or `` `emergency` ``, but this choice is made to create a complete "portfolio" for interprocess communication techniques.

Documentation and Makefile, what's under the hood?

Documentation

In this project all code is ~~well~~ documented with Javadoc style of Doxygen a standard tool for generating documentation from annotated code.

Notice: For Document compiling, doxygen is required

Mac OS (via brew)

```
brew install doxygen
```

Linux

```
sudo apt-get install doxygen
```

Then a useful Makefile is ready with a lot of commands included command for doc compiling.
After a `` `make doc` `` a web based documentation is available at `<project>/docs/html/index.html`

Makefile *[see file]*

For compiling over MacOS and Linux, makefile runs the shell command `` `uname` `` in order to determine if the underlying machine is linux based. It is necessary to link explicitly math libraries with `` `-lm` `` flag on Linux and exclude that flag on MacOS. A similar distinction is made on PFCDisconnectSwitch to be able to read status of process on Linux and MacOS (BSD).
In this project a command `` `make help` `` is useful to have a summary of capabilities of it.
To build and be able to run the project is required only to execute the command `` `make` ``

Tips: all folders like `/log` and `/obj` are created by hidden makefile command `` `make prepare` `` and they are deleted (except executable) by `` `make clean` ``