Method for Decoupled and Cohesive Data Communication in Avionics with Non-Terminating Threads

Brandon Mord

Computer Science

University of North Dakota

Grand Forks, ND 58201

brandon.mord@und.edu

Abstract

The purpose of this paper is to describe the development of software for the SmartSealz prototype that requires the incorporation of multiple source code languages to communicate data between non-terminating processes. A background of the work being done is provided along with an extensive description of the method used for creating a non-terminating process used in data collection. The goal of the development is to find a solution that allows for industry standards of cohesion, coupling, and scalability to be applied to a possible production-level prototype. The development of the communication method has the requirements of being accessible to all programming languages in use without the need for synchronization or causing delay of interface response for users. It also requires that the data collection be near real-time without the possibility of calling delay. Issues with other resources requiring data sharing protocols are considered including the use of database tools or shared memory protocols. The final method created is described in detail as using the file manipulation protocols accessible in every programming language. The use of this method is described with explanations of how it connects to industry standards of high cohesion, minimal coupling, and extreme scalability. Cohesion is measured by the level of required resources by each process in each of the different programming languages. Coupling is measured by the amount of connections the main interface process makes to collect the data in one location with restraints on process delay and data analysis methods. Scalability is measured by the ease of addition of similar tasks that require non-terminating processes in different languages used to communicate with micro-controllers. This file manipulation protocol creates an environment that most closely allows for high cohesion, low coupling, and extreme scalability in this avionics development domain. It allows for the transmission of data between the different programming languages in use now and in the future of prototype development.

1 Introduction

The use of avionics in the aviation world is vital to the success of flight. The SmartSealz development team consisting of two Computer Science undergraduates, two Electrical Engineering undergraduates, and two faculty advisors has been working on a standalone device that provides a touch screen interface as another form of avionics warning technology. The SmartSealz prototype takes the combination of avionic data gathered by self-contained sensors and can display the collected items on a touchscreen interface. This centralized interface is also the connection between the user and the different forms of feedback provided in the forms of visual warnings as well as vibrational, tactile feedback directly inside the audio headsets.

The need for some processes to be constantly running while data is transferred from this process to the user interface for display creates a special problem. The portion of the micro-controllers that gather any sort of physiological data or environment changes over time must be constantly running to provide current data and alerts upon changes in the environment. In the current prototype, there are three constantly running program scripts that require communication protocols to pass gathered data to the main interface.

The three scripts constantly running gather information to compare against specified bounds for error warnings. These three programs are not currently in the same programming language however, they need to provide data in its primitive type for a C source language program. The C source language program will then decipher and store data for comparison against deviation limits. The processes must never terminate and there cannot be any forced blocks or delays of functionality or the user will not receive the most accurate information.

The remainder of this paper looks at the strategy being used to ensure that the user receives the correct error feedback as well as how corrupted data items are removed without bothering the user. It shows how the method chosen maintains high levels of cohesion and scalability while decreasing coupling to provide clean and seamless information to the user.

2 Cohesion, Coupling, and Scalability Goals

The goal of the project pertaining to Cohesion and Coupling is the same idea used in the industry, the need for highly cohesive processes within their respective scopes and the decrease of coupling between the processes for data communication or information requirements for executions. The need to decouple the required resources that each process has from the other processes means the decrease of blocking calls or synchronization needs between information storage locations.

The pilot cannot input multitudes of information into each process, but the processes must be able to provide the user with all the information through one interface driven by the main source. This means that the main interface must rely on the processes that are running non-terminally to provide information through means of nonblocking methods. This also requires that the processes provide the information without receiving any information from the main interface.

The goal of Scalability is to allow for easy addition of similar functions and processes like that of industry standards. If the program is set up in a way that with the addition of new, similar functions and process, the amount of non-terminating processes can be increased in the same style as the initial non-terminating processes with limited coupling for the main interface source code.

3 Development Background

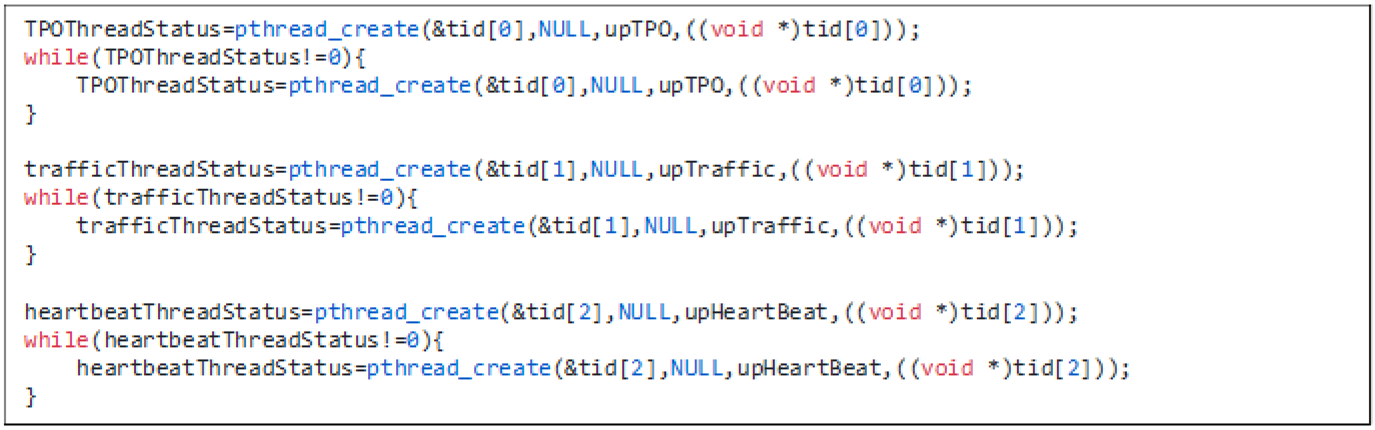
The development for the prototype has taken the initial concept to a flight functional prototype using a combination of different data collection micro-controllers. Currently, the composition of the prototype spans five different micro-controllers as well as processes that span six different languages including the Glade interface building tool set. This large range of languages has been developed through the ATOM IDE.

The touch screen interface using the GTK development libraries is driven by C source code programs and C libraries. The template for development was located at a forum providing the basic building blocks of a GTK interface [1]. This use of C creates the link between the actions performed by the user and the logic of displaying information, warnings, and sending feedback to the headsets.

The main challenge that this prototype has generated is the need for multiple languages to run different portions of the project. Due to the scope of this development project, development must allow for the addition of micro-controllers that may be designed for one specific language. The steps taken in development allows for the implementation of programs in these languages with relatively low integration effort but creates the special challenge of designing a single "plug-and-play" method of adding new data collection modules.

4 Execution Background

The current prototype has processes that need to be run in the background for the entirety of execution. This means they must be started and stopped from within a C source code program. This need for start and stop means that using the C "system()" command is not a plausible solution alone because of the blocking nature of "system()". The use of the "exec()" suite of commands is also not the best option for keeping track of the child threads because there is the issue of permissions on global variables and increases the need for shared memory on a system with limited resources. The solution is using the built in "pthreads" library to spawn child functions by running the "system()" command from an unblocked thread.

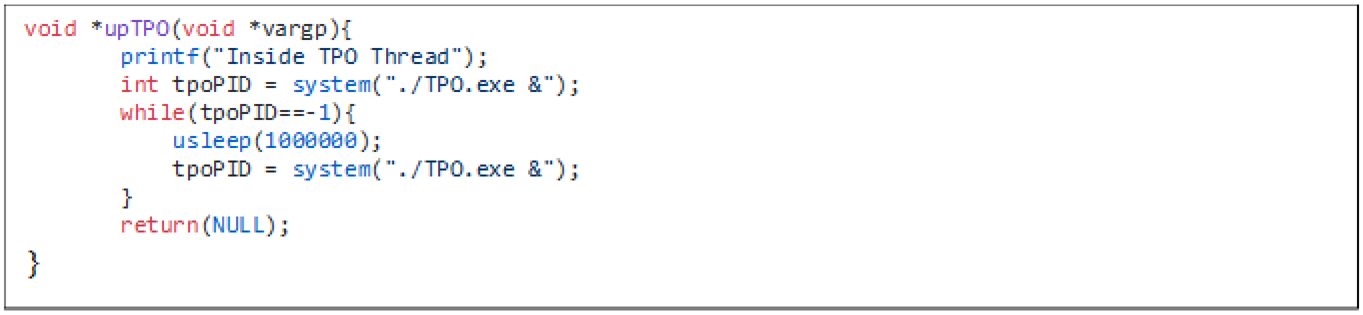


```
TPOThreadStatus=pthread_create(&tid[0],NULL,upTPO,((void *)tid[0]));
while(TPOThreadStatus!=0){
    TPOThreadStatus=pthread_create(&tid[0],NULL,upTPO,((void *)tid[0]));
}

trafficThreadStatus=pthread_create(&tid[1],NULL,upTraffic,((void *)tid[1]));
while(trafficThreadStatus!=0){
    trafficThreadStatus=pthread_create(&tid[1],NULL,upTraffic,((void *)tid[1]));
}

heartbeatThreadStatus=pthread_create(&tid[2],NULL,upHeartBeat,((void *)tid[2]));
while(heartbeatThreadStatus!=0){
    heartbeatThreadStatus=pthread_create(&tid[2],NULL,upHeartBeat,((void *)tid[2]));
}
```

Figure 1: Thread Spawn Code



```c
void *upTPO(void *vargp){
        printf("Inside TPO Thread");
        int tpoPID = system("./TPO.exe &");
        while(tpoPID==-1){
            usleep(1000000);
            tpoPID = system("./TPO.exe &");
        }
        return(NULL);
}
```

Figure 2: Non-Terminating Executable Start Thread Code

Figure 1 shows the process of spawning the "pthreads" in conjunction with Figure 2 which gives an example of one of the threads that is being used to start a non-stop executable file. Since the "system()" command blocks until termination and the processes being run by these commands are scripts that are non-terminating, the threads are a safe alternative and allow for the processes to be started and stopped forcefully within the same program. The only thing that must be provided to stop these threads are the thread ID's that can be gathered in each individual script.

Each process that is started as a non-terminating system call gets the thread ID that it is currently being run under and prints it to a file for the main interface source code to collect after the creation of the threads. This allows for, upon user termination, the threads to be forcefully stopped such that the system releases all allocated space and doesn't leave behind orphaned child processes.

This method of "processing spawning" allows for a high level of cohesion in the functions. Each process must have all the information present within its files to run properly and is not delayed by user input or API calls. This solution also increases coupling between the files holding the thread IDs and the main source, but it allows for the decoupling of functions that are controlled by programs in other languages. The processes are contained within their respective languages and do not rely on any form of start and stop signals from the user. This however created the issue of how the to communicate data between the separate processes containing different language syntax for primitive types.

5 Communication Problems

Since the user interface is controlled by C source code there are not many built in libraries that allow for the use of JSON data communications. This is also accompanied by the remaining issue of constantly running processes needing to connect to the C source code program through a similar method of connection. Since we are trying to send large amounts of data from different sources, there is the need to have a mapping system, so data is not lost in transition between languages.

One option considered was connecting directly with the serial protocol. The main issue with this protocol is the large amounts of data that each process must collect and relay back to the main interface program. Since this could potentially cause delays with any files transmitting large amounts of data, this method could delay data reading processes. Because of this potential for delay, this protocol is not sufficient for this application.

Another protocol considered was database calls using SQL queries, but this was quickly dismissed due to the need for constant processes running continuous updates on the tables. These updates could lead to possible data collisions as other processes attempt to read from the tables. The use of a database is still being considered for the possible need for recovery storage, but it is not something that can be used at these relatively high refresh rates without the worry of loss of data.

The main issue was the need to decouple the processes. The main interface program needed to be able to receive all the information in one section of code and continue without extreme delays. It couldn't be dependent on each process being in a state to transfer data directly or wait for each process to have their turn with the database and wait on writing the next refresh until every other process has finished its write.

The goal was to find something like shared memory but that would not have to be allocated and managed by a taxing process and wouldn't require multiple coupled points each causing blocking in the separate processes. It also needed to be easily updated as well as removable if specified by the user. It had to have the ability to hold string, integer, and float data types. It had to easily overwrite itself as the new data was provided. And lastly, there had to be conventions in each language that would allow for this "middleware" to be a universally understood protocol.

After some consideration, the use of flat file operations was the decided to be the best solution. Every language can write out to a file and read in information from the files. The data is easy to convert from string to other data types with the "atoi()" and "atol()" functions. The files are also easily overwritten, created, and deleted throughout the program. This also allows for each process to be coupled to its own file for data transfer purposes, so no blocking calls needed to be made or synchronized between the processes.

Some of the issues that can arise from this style of data transmission are lost read and write problems. Since the processes writing to the files are not being delayed in any form and the process reading the data from these files is doing it on a regular interval inside a loop, there have been issues with double access of the file at the same time. This issue was discussed and if the information the user sees is refreshed four times/second the device is defined as preforming adequately.

The current prototype is running at ten data refreshes per second and the user will never receive any unwanted feedback in the event of lost data because the warning flags are checked against the most current correct data available. The feedback that the user sees and feels comes from one of at least four readings every second but can range anywhere between four and ten times/second.

```
void *readFile(void * arg){
       while(1){
    inFile = fopen("DataFiles/Nav.txt","r");//Navigation data file
    if(!inFile){
               printf("Ignore Information\n");
    }
    else{
        fscanf(inFile,"%d %f %d %d",&inAlt,&inPres,&inPitch,&inRoll);//Read in Navigation Data
        fclose(inFile);
    }
    pthread_mutex_unlock(&readNavLock);
    pthread_cond_wait(&readNavCond,&readNavLock);
  }
  return (NULL);
}
```

Figure 3: Read File Thread Code

Another issue that arose was the increased coupling of the main source to these four files that are being generated and constantly written and overwritten. This was addressed with the idea of creating another set of threads like Figure 3 that will attempt to open the file. If the file cannot be opened or the data is improperly formatted the threads return with the previous data to be re-displayed. If they can read the file they will replace the information and prepare the display for new information. Each file is given a thread so that there are three files being read simultaneously and the information is being consolidated at the pre-display phase.

This increased coupling however is not a full link between the main source and the files because the files can act as an endpoint that a database or API cannot. The database or API will return a status of the call where the reading of the files allows for all the lifting to be incorporated into the calling side. If the data is seen as being corrupt, then the calling side can move on without being blocked and waiting long lengths of time for any status returns.

6 Current Solution

The current solution that is being used in the prototype allows for the program to call the four created files all from separate threads and collect all the information before the display phase. This connection solution allows the program to access these four folders multiple times a second, read the information, determine if the information is corrupt or valid, then stores the information or continues with the previous information.

The current solution also keeps track of the ID numbers for the non-terminating threads. These files are created on start of the threads, accessed by the final cancellation function, and removed by the cancellation function to remove the unnecessary files and free up disk space. These files are relatively safe from any data corruption because they are only written too once and read from once. The main reason these transfers follow the same protocol is to allow for scalability and provide an example for how the removal of the files can be handled.

Each of the processes get their startup system call, preform their necessary actions in a loop to collect the data required, print the data to the respective file, then close the file to allow for file reading operations in a different program. This ensures that the data being read is relatively real time and any issues reading the files can be handled on the software side versus requiring the user to be aware of the issues and possible handling issues.

7 Conclusion

The development process has provided an interesting challenge for the transmission of data across different languages with the idea of increased cohesion and decreased coupling. The method for this data communication needed to be a scalable option that could allow for increased numbers of non-terminating threads that provide information for the user.

Using the different built-in libraries and easily accessible resources across all languages, the use of file writing and reading protocols were a valid solution to the problem being analyzed. The files allow for the easy creation, overwriting, and deletion of information needed by the main interface source code. This method of data transfer also easily allows for different language primitive types to be sent and interpreted by the main interface source code.

Creating a cohesive environment for each of the processes was achieved using the idea that the different languages will not rely on any sort of information to execute data collections. The processes are started separately and collect data autonomously. They can execute their functions without any sort of interrupts or any blocking calls.

Creating a decoupled environment is achieved by placing the data integrity analysis in the main interface source code, allowing the data collection to be done real time and doesn’t block the main interface. The processes providing a single file with the information allows for the main interface to pick up the information, analyze, and store or remove the collected data. Decoupling occurs because there is no requirement for multi-connection data transmission or blocking synchronization measures.

The use of the file transition method has provided the most significant results for a near real-time system that has little intervention from users. The functionality of the software must be constantly considered with possible innovations in both hardware and software creating new solutions. The level of scalability provided by this method allows for possible software additions and continued development.

[1] CodeNerd, Author. “GTK 3 Glade C Programming Template Files.” Programmer's Notes, 5 July 2016, https://prognotes.net/2015/07/gtk-3-glade-c-programming-template/.