CHALLENGES IN EUROPEAN AEROSPACE
5TH CEAS AIR & SPACE CONFERENCE

CEAS

A distributed simulation framework for rapid deployment of research demonstrators

Ricardo M. U. Entz

Airbus Group Innovations

PhD Candidate

81663, Munich, Bavaria, Germany

ricardo.entz@airbus.com

Emilio E. T. Moreira, Rafael Fernandes de Oliveira, Falk Schubert, Dirk Dickmanns, Erwin Stenzel, Gabriel Scherer Schwening

Airbus Group Innovations

ABSTRACT

In aeronautical research the scientist often faces the challenge of demonstrating their results in a way that is more user friendly than tables and charts. In Airbus Group Innovations (AGI), demonstrating disruptive technologies starting from a very low technology readiness level is essential to confirm customer interest and gather feedback about how they can be further developed. As a result, several lightweight, *ad-hoc* tools are developed to provide visualization, human interaction and software in the loop capable demonstrators – a non-value adding task that consumes time that would be better used in the technologies being developed.

This scenario led us to identify the need to adopt a unified simulation framework for rapid prototyping of simulations in a research environment. In this work, we describe the result of this development, the main design decisions and some example applications.

The initial envisaged applications for such a framework were the integration of trajectory optimization, weather radar and computer vision activities developed within AGI.

1 INTRODUCTION

1.1 Motivation

One of the most used technology assessment tools - Technology Readiness Level (TRL) - require the demonstration of systems in environments of increasing fidelity, rendering a flight simulator an indispensable tool in the maturing of a technology of intermediate maturity (TRL 4 to TRL 5).

The Autonomous Systems and Image Processing team at Airbus Group Innovations has a tradition in developing technology bricks that perform certain functions within systems with various degrees of autonomy. Examples of these tasks involve image-based landing, obstacle detection and trajectory optimization. While flight simulation is an essential part of the development cycle of these functions, the implementation of a simulation environment to test them is usually far beyond their scope.

In this context, a common modular simulation framework solution was envisaged to allow early testing of new technologies with the minimum effort possible, so that flight simulation can be performed in an early development stage, for development and demonstration purposes.

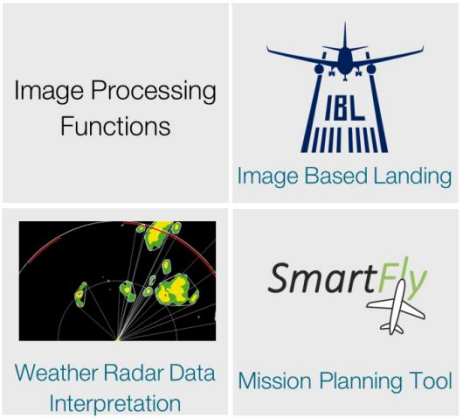

Figure 1 - Autonomous Systems and Image Processing team technology bricks

1.2 Requirements

As a first step in the simulation framework definition, a requirement document was created taking into account the diversity of technologies, fields of research and software development philosophies existing in the Autonomous Systems and Image Processing team. Each principle will be presented in this section.

The framework shall be capable of supporting simulations of a varying degree of fidelity

Having a pool of technologies at different maturity levels demands a simulation environment capable of following their development pace. A simulation that is too elaborate can be burdensome to be applied to a proof of concept and a simulation that is limited becomes useless in later development stages.

There must exist programming language-agnostic means of interfacing with the simulation

In a research ambient with so much diversity in subjects, maturity levels and project sizes, several programming languages coexist. Imposing a language to interface with the simulation would only reduce the motivation of a new user to adopt the proposed solution. Ideally, a text protocol using a commonly available interface would be ideal.

The framework must be scalable

For larger simulations, it is essential for the framework to be capable of executing simulations distributed over several machines. This also allows for hardware-in-the-loop (HIL) demonstrations.

The framework must provide flight simulation tools, but be simulation agnostic

While our main application is flight simulation, it is important to keep the framework as generic as possible. Having flight simulation tools is useful to kick-start a demonstrator, but we have to provide for users not interested in flight simulation. Tightly coupling the development to a flight simulation would only create showstoppers for such users.

The framework source code should be available and distributable

While the framework is to be treated as a black box by the user, it is important to be able to access the source code and, if necessary, distribute it without licensing problems. In addition, with thoughtful code maintenance, the code can be ported to several platforms.

The data should be human-readable

Most solutions trade human-readability of its data for performance. However, if we take a simple flight simulation as an example, a few dozen data fields need to be exchanged at each simulation step. This led us to accept that the overhead of converting data between human and machine readable formats is negligible compared to the improved clarity when testing and debugging a simulation.

1.3 Off-the-shelf solutions

After the requirements document was elaborated, a search for existing solutions was made. Some requirements, however, narrowed the search space considerably. One example is that only open-source solutions were considered in order to guarantee availability of the source code at a reasonable cost. Therefore very few mature projects were found, the most noteworthy solutions are listed in this section.

```
<function name="aero/coefficient/Clr">
    <description>Roll moment due to yaw rate</description>
    <product>
        <property>aero/qbar-area</property>
        <property>metrics/bw-ft</property>
        <property>aero/bi2vel</property>
        <property>velocities/r-aero-rad_sec</property>
        <table>
            <independentVar>aero/alpha-rad</independentVar>
            <tableData>
                0.000 0.08
                0.094 0.19
            </tableData>
        </table>
    </product>
</function>
```

Figure 2 - JSBSim example - definition of a property value as a function of other nodes. This kind of program flexibility was a main inspiration for DSF

OpenEaagles [1] is a simulation framework compliant with the DIS (Distributed Interactive Simulation) standard [2] [3] [4]. DIS specifies a communication protocol used for transmission of battlefield entities state. A problem with this solution is its specialization on defense applications, so a general-purpose use would require some adaptation, and disabling of some features. While this is feasible, we concluded that the adoption of DIS would bring a too steep learning curve that would reduce the potential user base of our solution.

JSBSim [5] is a flexible, open-source flight dynamics model. This solution was extensively studied during the initial phases of the project despite its limitations in complying with the requirements, namely on the language and simulation independence. The reason for this in-depth study was its interesting architecture, based on a data tree configurable with XML files. The system is created in a way that the values can be set by external programs or by internal calculations defined by the user, as shown in Figure 2.

FlightGear is an open-source flight simulator with a large user base. This solution was also studied as inspiration because of its modular architecture, with a data tree that can be manipulated both by internal and external modules [6]. In the case of external models, the manipulation of the data can be done through a network interface allowing for some distribution of the code execution. In the other hand, the licensing model of FlightGear is a problem. It is licensed under the GPL (General Public License) [7], which requires any statically-linked code to be available to the public under the same license.

After a study of these solutions, it was decided that a custom-made, lightweight solution inspired by the flexibility and modularity of JSBSim and FlightGear would be the ideal solution for our use case.

2 DEVELOPMENT OF THE DISTRIBUTED SIMULATION FRAMEWORK

This section shows the main design decisions taken in the development of the AGI Distributed Simulation Framework (DSF).

2.1 Definitions

Before we go further with the definition of the DSF, it is important to clearly define some terms that will be used in this work.

DSF-Server is the central entity of the simulation, responsible for gathering data, transmitting it and ensuring synchronization. In some simulation cases the DSF-Server is also responsible for synchronization between simulation time and wall clock.

Client(s) are simulation elements (e.g. flight dynamics, rendering, autopilot) that need to exchange data. These elements are developed and executed independently of DSF-Server, as shown in Figure 3.

In the text, the word Client will be capitalized to designate DSF-Clients, i.e. programs that adhere to the DSF protocol.

Property Tree is the structure where the exchanged data is stored. This structure is internal to the DSF-Server, but is accessible in a read-only mode to all Clients. Some nodes (data fields) in the Property Tree are owned by Clients. A Client can only write to a node it owns.

2.2 General development aspects

The simulation is defined as a continuous data exchange among Clients using the DSF-Server. This way, the main functional requirements for the program are:

1. Be able to send and receive messages;

2. Be able to collect the data from the messages and organize it according to the needs of the clients;

3. Be able to synchronize the messages.

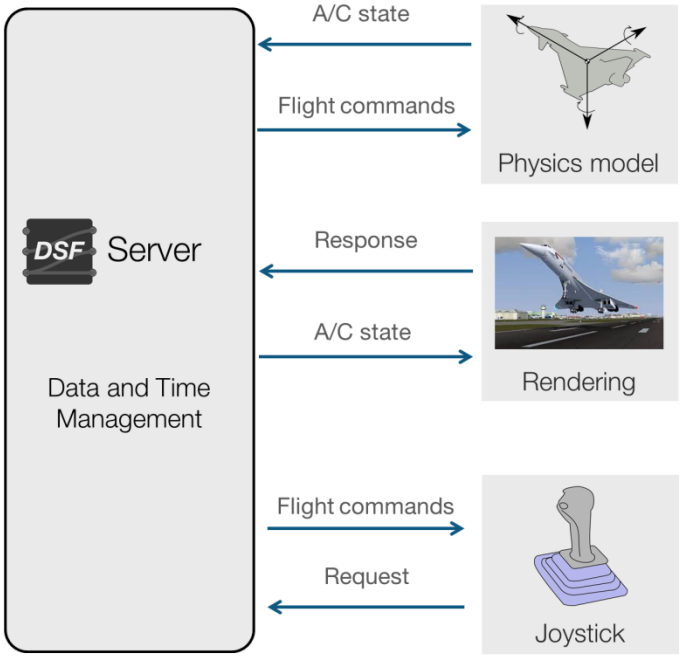

Figure 3 - Example of simulation architecture with data exchange. In DSF each client (Physics model, Rendering, Joystick) are separated processes, for enhanced modularity.

Some development guidelines were defined inspired by "The Zen of Python" [8] as an effort to create a flexible, lightweight solution. The most important points are listed below:

1. *Readability counts* - all data is kept as human readable as possible, with extensive use of XML files and the exchange of text-based messages;

2. *Special cases aren't special enough to break the rules* - we tried to find an operation logic that allows the user to implement special cases without changing the core software. This way the DSF-Server can be seen by the user as a black box;

3. *Errors should never pass silently* - whenever possible, sanity checks are performed and the simulation is stopped if not compliant; and

4. *In the face of ambiguity, refuse the temptation to guess* - all options are externalized, the architecture is extensible and decisions on the simulation architecture can be made by the user.

As the performance of the DSF-Server was critical for its usefulness, we decided to use C++ as the main programming language of the project. This allows the server to have a maximum refresh rate in the order of thousands of Hertz with wall clock synchronization, an important criterion for the execution of real-time simulations.

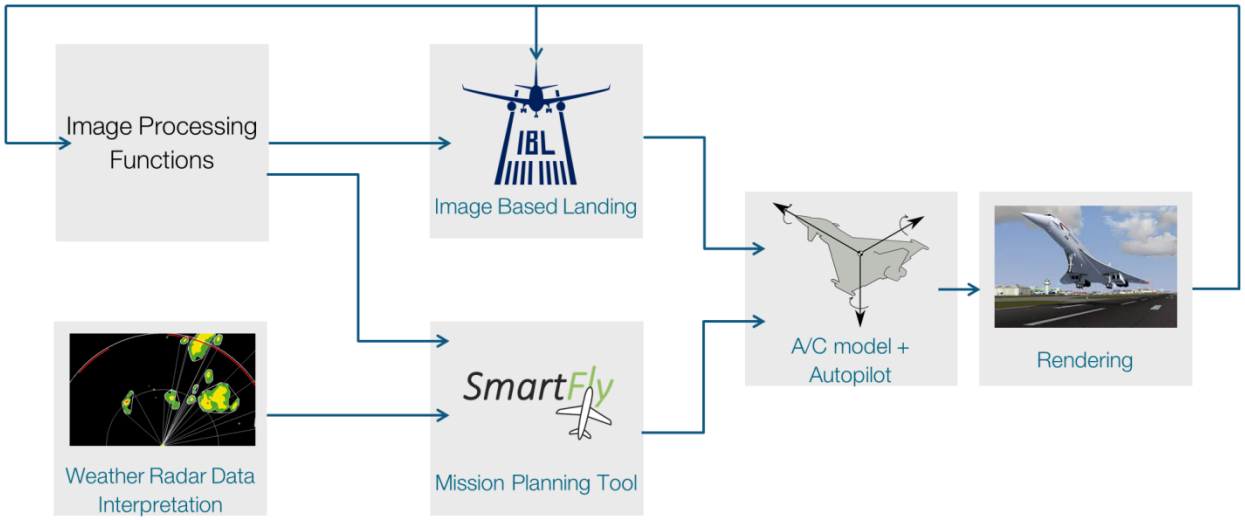

Figure 4 - Vision of additional technology bricks and integration provided by the simulation framework

2.3 Data model

In the DSF-Server, the central part of the simulation is the Property Tree, where all data is stored and retrieved. In this structure, data is stored as plain strings - a logical decision if we consider that the DSF-Server sends and receives messages in plain text and performs no calculation with the data, only routing. This also enhances data readability and flexibility of the program, as required by points 1 and 4 of the general concepts.

Some obvious drawbacks of this decision are the performance cost of parsing, converting data and its associated risks, such as wrong conversions. This performance penalty can be a problem if a lot of data needs to be exchanged at high frequencies.

For most simulation cases, however, there is not much data being exchanged between simulation entities - e.g. an aircraft state can be determined with six floating point numbers (position and orientation) and the most common simulation parameters make up for a few dozen parameters only (engine state, control

surface positions, etc.). In addition, the parsing cost is left entirely to the Client; the parsing/conversion burden is spread among several agents of the simulation, over several machines and processors.

2.4 Communication

DSF-Server adopted network sockets as the model to communicate with its Clients to provide cross-language, platform and machine capabilities. This approach was inspired by FlightGear and JSBSim, that provide access to their Property Trees using Ethernet.

In our implementation, we current have TCP and UDP capabilities over the socket interface. We left the possibility of implementing other protocols and interfaces by using a strategy pattern [9].

This approach goes in line with the design philosophy, but it is seemingly incompatible with the concept of real-time simulation present in the requirements document. This is due to the fact that most Ethernet protocols (notably TCP and UDP) do not have real time characteristics, due mostly to the routing and switching of packages. There is extensive research on real-time communication protocols over Ethernet [9], some even developed within Airbus Group [10], but they require communication at a too low level to allow simulation deployment without the use of specific libraries.

The core code of the DSF-Server should ideally not be dependent on the details of implementations of low level communication protocols, first because we want a lightweight solution and second because the implementation of a new protocol would bind the clients to a "DSF-Library", that is not desirable.

As an alternative solution to this problem, we adopted a transport scheme consisting of a "buffer and update" cycle that provides for slack time in the communication, and processes the messages in a deterministic way. Our proposal is described by the following algorithm:

1. The DSF-Server message is generated with a timestamp;

2. The message is sent to the target Client;

3. The DSF-Server expects to receive an answer from the Client in a given amount of time, defined as *Time Frame*;

4. The DSF-Server receives the answer anytime between sending it out and the end of the Time Frame, the *deadline*;

5. When the DSF-Server receives the answer, it performs a check of the content, parses it and stores the data in a buffer;

5. If the message contains information for the Property Tree, this information is only updated at the end of the Time Frame;

6. If no message is received until the deadline the DSF-Server will either stop the simulation or issue a warning message to the user.

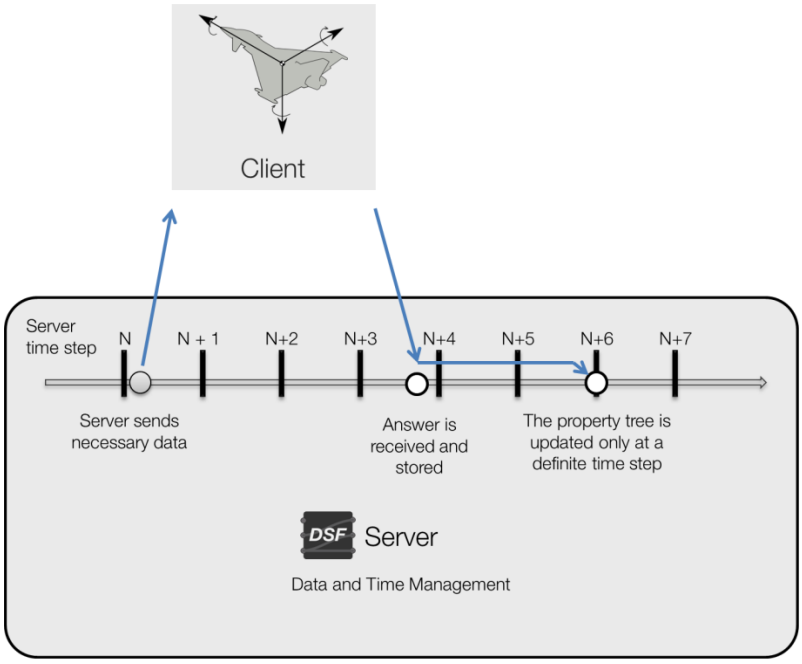

Figure 5 - Buffer and update strategy of DSF-Server. Here the Time Frame has a length of 6 server time steps, the message can arrive anytime between time steps N and N+6, but the Property Tree will only be updated at time step N+6.

In order for the scheme to work, the Time Frame duration has to be set to more than the estimated worst-case message round trip plus the Client processing time. From this point of view this solution is extremely wasteful, on the other hand it is simple to understand and implement on the client side, which can use standard socket libraries.

Another drawback of using such approach is that all the system is forced to be executed at a lower refresh rate than the DSF-Server and clients maximum, to provide the necessary slack for delays in the communication protocols.

After implementing a proof of concept of the "buffer and update" cycle, we considered that the achievable communication frequency was acceptable. For example, a typical AGI simulation has a DSF-

Server time step of 1ms (1000Hz) and the Image Rendering Client is refreshed each 10 time steps, resulting in a final refresh rate of 100Hz.

2.5 Time management

The DSF-Server is capable of executing simulations in three different modes, namely:

1. Hard real time;

2. Soft real time; and

3. As fast as possible.

Real time modes are synchronized to the wall clock, so that human-in-the-loop and hardware-in-the-loop simulations can be executed. In the hard real time mode the simulation is halted if a deadline is missed by any of the clients.

The third mode, as fast as possible, delegates the simulation pace to the clients. In this mode the DSF-Server waits until all scheduled messages are received. This is useful for debugging a simulation and also in cases where keeping synchronization with the wall clock is not essential, for example when a very slow client is being used.

3 SETTING A SIMULATION UP

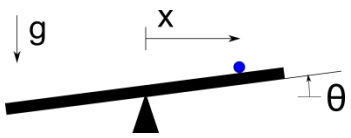

Figure 6 - Simple physical model used as an example DSF application

This section will show how a simulation can be set up within DSF. The objective is to demonstrate in practice some of the details of how the DSF can be used.

For this section a very simple physical system consisting of a sphere that can move freely in a tiltable rail, as shown in Figure 6, will be simulated.

In this problem we will control the angle, θ, to keep the sphere at the center of the rail ($x = 0$). The system can be described by the equation:

$$\ddot{x} = -g \cdot \sin(\theta) \tag{1}$$

The simulation will consist of three different Clients:

1. The dynamic model, based on Equation (1);

2. A PID controller, that acts on the angle θ;

3. A visualization utility.

In order to set-up the DSF-Server, it is necessary to determine which variables will be passed from client to client. These are shown in Table 1.

Table 1 - Variables exchange between simulation entities in the example problem. Blank cells indicate that parameter is not used by entity.

	DSF-Server	Dynamic model	Controller	Visualization
Sim. time	*Output*	*Input*	*Input*	*Input*
x		*Output*	*Input*	*Input*
$\dot{x}$		*Output*	*Input*	
θ		*Output*	*Output*	*Input*

On the synchronization side it is necessary to determine how much time a client needs to perform its calculations and how long a message takes to be sent. As a simplification, we will assume that a message can be passed in less than 2ms and the clients take the following time to process the data:

1. Dynamics: 2ms

2. Controller: 2ms

3. Rendering: 10ms

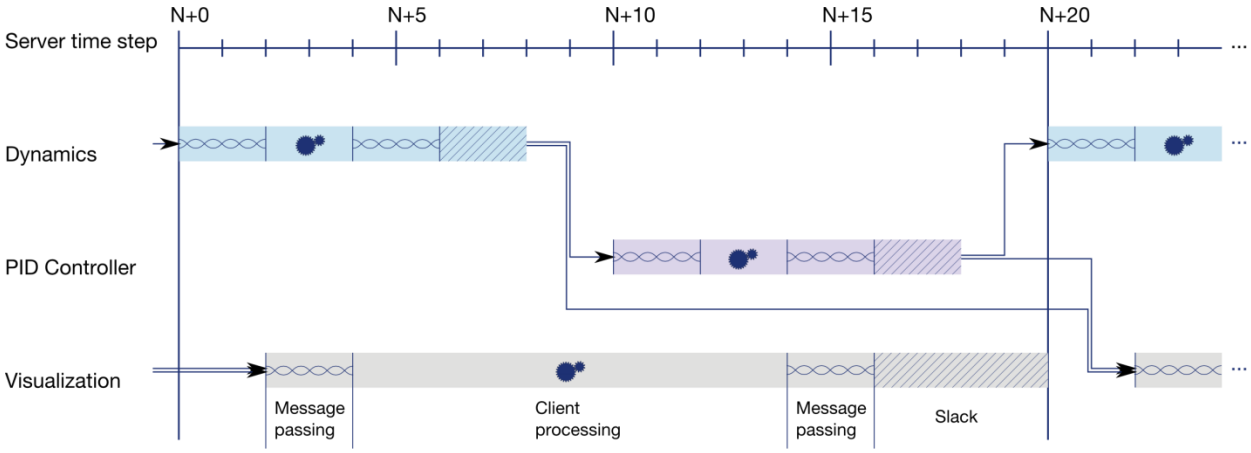

Figure 7 - Synchronization of clients in the example. Server time step is 1ms.

With this information we create a scheduling as shown in Figure 7. The figure also breaks down each Client Time Frame in four tasks: receiving message, processing, sending message and slack.

It is important to notice that the DSF-Server expects an answer from all Clients, including the Visualization, that has no outputs. In this case the answer contains just an acknowledgement of reception to determine if the client is still performing its task, reducing the risk of executing invalid simulations.

From the figure it is possible to define important values needed to fully determine the simulation synchronization:

1. **Time Frame duration** is the time the Client has to answer the message. The end of the Time Frame is the Client *deadline*.

2. **Offset** from the N+0 time step. This value is used to establish the order that the clients will be polled. In our case the PID Controller is always polled after the Dynamics Client.

3. **Frequency** that each client is polled. In this case we work with the same frequency for all the Clients (polling each 20 time steps, 50Hz) for pure convenience; the DSF-Server is capable of working with different frequencies at the same time to allow integration between fast and slow Clients.

With the simulation synchronization defined, it is possible to describe how the execution of this simulation would happen.

3.1 Simulation execution

When the DSF-Server is launched, it reads a XML file with some basic configuration and then waits for Client registration, staying in a mode denominated *paused*.

The registration is done with a XML file denominated *Contract*. This is sent through a socket via Client connection and it contains the following information:

1. Client name;

2. Nodes that Client will write in the DSF Property Tree (i.e. Client outputs);

3. Initial conditions of these nodes;

4. Nodes from where Client will read in the DSF Property Tree (i.e. client inputs);

5. Frequency that the Client will be polled (given as a period, in time steps);

6. Duration of the Time Frame; and

7. Offset of the time frame.

In the paused mode the DSF-Server also accepts commands to edit the Property Tree, adding, removing or editing nodes. It is also possible to dump the Property Tree to check its values. A web interface is provided for that.

After all clients are registered, the user can request the simulation to start, taking the server from the paused to the running mode. This triggers the following procedures:

..3.1.1 Property tree consistency checking

The DSF-Server ensures that no inputs are left undefined. As the Clients can register asynchronously when the DSF-Server is in paused mode, the Property Tree can only be checked at the beginning of the simulation. This check ensures that all nodes that are read by the Clients exist.

Another condition that is checked is whether a node is edited by no more than one Client, to ensure data integrity in the simulation.

In our example, the Property Tree has the following nodes after all clients register:

• /Native/Simulation_Time_us

• /State/x_m

• /State/x_dot_ms

• /State/theta_rad

Note that the DSF-Server automatically generates some nodes in the Property Tree in the folder "Native" such as simulation time and time step.

..3.1.2 Client initialization

There is a missing point in the simulation - the initial conditions. In the last section, we have seen that the contract defines initial conditions, but they apply to a Clients' outputs, not inputs. Before all Clients are registered, the DSF-Server does not know the relation between Clients; on top of that, the user can edit the Property Tree using the DSF-Server Interface. As a consequence, the initial conditions of the simulation are unpredictable.

We address this problem using an optional initialization scheme performed right before the beginning of the simulation. It consists of the following steps:

1. The user launches a simulation;

2. The DSF-Server leaves the paused state, preventing user changes in the Property Tree and registering of new clients;

3. The Property Tree consistency is checked, to determine if all Clients have the necessary inputs;

4. Clients that opted to be initialized receive a message with the necessary data;

5. When all these Clients acknowledge the receipt of the initialization message, the simulation begins.

There is a deadline for the Clients to answer to the initialization message, but it is much longer than the normal simulation deadline (in the order of seconds, but configurable) as some Clients may have slow initialization.

..3.1.3 Execution

This section shows how the DSF-Server executes our example simulation. The first 20 time steps are necessary to complete a whole simulation cycle.

Time step 0 a message is sent to the Dynamics client with $theta\_rad$ and $Simulation\_Time\_us$.

Time step 2 a message is sent to the Visualization model with the values of x_m and $theta\_rad$ stored in the Property Tree.

Between time step 0 and 8 the DSF-Server receives a message from the Dynamics client with the values of x_m and x_dot_ms. These values are stored in a buffer.

Time step 8 the buffered values of x_m and x_dot_ms are stored in the Property Tree. If no message was received until now an error is thrown.

Time step 10 a message with the values of x_m, x_dot_ms and $Simulation\_Time\_us$ is sent to the PID Controller Client.

Between time step 10 and 18 the DSF-Server receives a message from the PID Controller model with the value of $theta\_rad$. This value is stored in a buffer.

Time step 18 the buffered value of $theta\_rad$ is stored in the Property Tree. If no message was received until now an error is thrown.

Between time step 2 and 20 the DSF-Server receives a message from the Visualization model indicating that the rendering was successful. If the message is not received by time step 20 an error is thrown.

..3.1.4 Stopping execution

In order to guarantee data consistency the DSF-Server only stops execution when there is no data pending to be saved to the Property Tree. For our example this happens every 20 time steps, this is represented by the solid line in Figure 7 and denominated *measure*.

When the simulation is stopped, the DSF-Server returns to the paused mode. In this mode new clients can connect and the Property Tree can be externally edited.

4 STANDARD DSF LIBRARY

The DSF-Server by itself is a powerful tool for simulation deployment, but a solid library is an essential tool to expand the user base, as it can reduce the cost and increase the benefit of adoption.

The library components are implemented as DSF- Clients, for maximum flexibility. This section lists some of the components of the standard DSF library.

4.1 FlightGear client

FlightGear Flight Simulator (FGFS) [11] is used in the DSF as the standard renderer, a natural choice in the flight simulation field as it is capable of generating high quality images of aircraft and terrain; it is extremely customizable and has a large user base.

It is not possible to link the DSF code with FGFS due to its strict licensing terms, so an interface that adapts the DSF protocol to that of FGFS was created. This interface takes advantage of the command line and UDP socket interfaces provided by FGFS, this allows DSF to control aircraft position and orientation, weather and add other models in the simulation environment.

4.2 Simulink client

Simulink is a widely adopted software for simulation of dynamic systems. In order to allow seamless integration with DSF a Simulink block was created. It is capable of automatically generating a XML contract, based on its inputs and outputs.

4.3 Human interaction devices client

For human-in-the-loop simulations a customizable interface was created. It is capable of reading multiple joystick inputs, communicating their state to the DSF-Server.

5 EXAMPLE APPLICATIONS

This section shows some demonstrators already developed within AGI using the DSF.

5.1 Displaying optimized trajectories with a tunnel in the sky visualization

Aircraft trajectory optimization has been a major research topic of the AGI Autonomous Systems and Image Processing team [12] [13]. In addition to full flight simulations used to bring the technology to a higher TRL, the DSF is now a major component of the trajectory optimization suite for visualization purposes.

Optimized trajectories can have elaborate, coupled maneuvers, and it is important to have an intuitive way of communicating these procedures to pilots, both in briefings and on-board. For this purpose, we adopted the tunnel in the sky approach.

In this application the DSF-Server integrates the mission planning tool, FlightGear and a tool to position the elements of the tunnel (added as extra models in FGFS). Figure 8 shows an example of plotted trajectory.

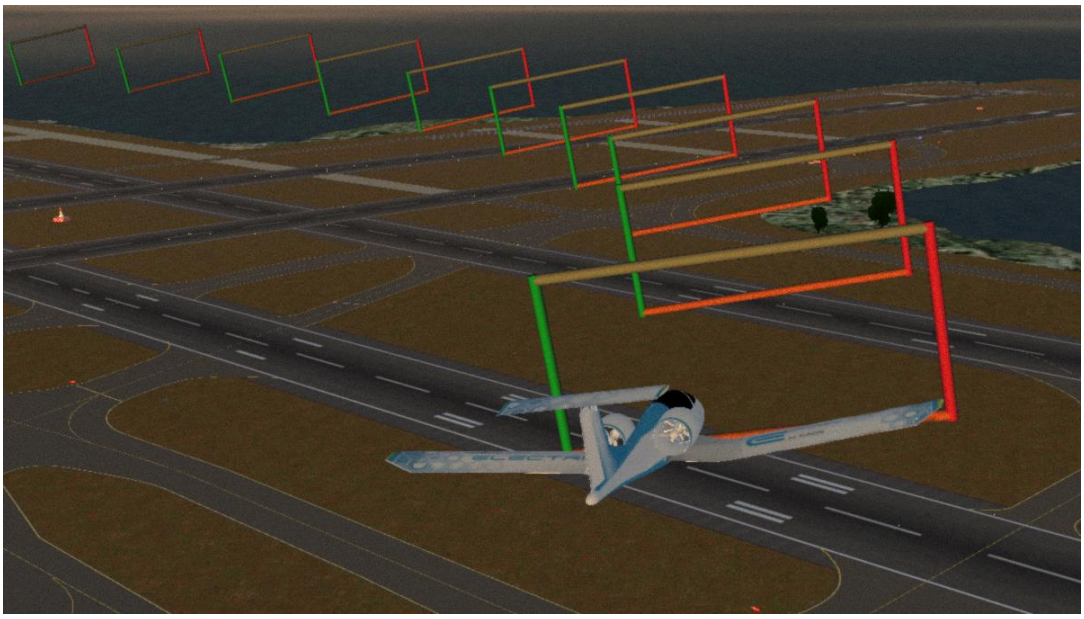

Figure 8 - Example of tunnel in the sky as a means of visualization of optimized trajectories

5.2 Application of image processing applied to air-to-air refueling

The development of tracking algorithms demands a thorough verification and validation. The problem in aeronautical systems is that high fidelity tests are too expensive to be performed; this creates a potential application for our Distributed Simulation Framework.

This simulation aims to represent a tanker aircraft with a hose and drogue refueling system. The hose has a very complex behavior depending on its inertia, the aircraft movement and the aerodynamic interaction between both.

We developed a DSF-Client capable of taking as input the tanker state and computing the hose shape. This was integrated with other clients via DSF as shown in Figure 9.

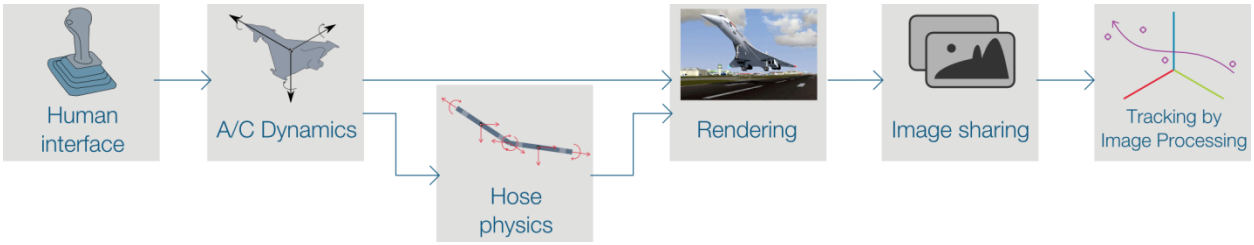

Figure 9 - Simulation chain for air-to-air refueling simulation

The connection between FGFS (our rendering engine) and the image processing algorithms was performed through a custom-build Image Sharing module allowing high frequency image capture (30+Hz) and online tracking of the drogue by image processing, as shown in Figure 10.

Figure 10 - Drogue tracking in a DSF simulated environment

6 CONCLUSION

This work presented the development and application of the Distributed Simulation Framework (DSF), a tool to simplify the deployment of research demonstrators by providing a modular solution to simulation.

With the tool starting to be adopted by its first beta testers, it was possible to conclude that its success is due to some major factors - its simplicity, flexibility and standard library. By not binding the user to a

programming language or a simulation architecture the adoption cost was kept to a minimum and its standard library boosts its benefit.

The future work on the tool aims to improve its user-friendliness with respect to simulation management (e.g. improving the server web interface), an essential step towards bringing the DSF to more widespread use.

7 REFERENCES

[1] D. Hodson, "OPENEAAGLES, An Open Source Simulation Framework," *A Publication of the AIAA Modeling and Simulation Technical Committee,* vol. 1, no. 1, 2008.

[2] IEEE, "1278.1-2012 - IEEE Standard for Distributed Interactive Simulation - Application Protocols," IEEE, 2012.

[3] R. C. Hofer and M. L. Loper, "DIS today (Distributed interactive simulation)," *Proceedings of the IEEE,* vol. 83, no. 8, pp. 1124-1137, 1995.

[4] M. McCall, "DIS 101 Tutorial," Simulation Interoperability Standards Organization, 2013.

[5] J. S. Berndt and A. De Marco, "Progress on and usage of the open source flight dynamics model software library, JSBSim," in *Proceedings of the 2009 AIAA Modeling and Simulation Technologies Conference*, Chicago, 2009.

[6] E. F. Sorton and S. Hammaker, "Simulated flight testing of an autonomous unmanned aerial vehicle using FlightGear," *American Institute of Aeronautics and Astronautics, AIAA 2005,* vol. 7083, 2005.

[7] "GNU General Public License," Free Software Foundation, 2007.

[8] T. Peters, "The Zen of Python," Python Software Foundation, 2004.

[9] F. T. Y. Hanssen and P. G. Jansen, "Real-time communication protocols: an overview," Centre for

Telematics and Information Technology University of Twente, Enschede, 2003.

[10] J.-P. d. Moreaux, "Data transmission system for aircraft". Germany Patent DE1999154377 19991112, 02 08 2005.

[11] A. R. Perry, "The FlightGear Flight Simulator," in *Proceedings of the USENIX Annual Technical Conference*, 2004.

[12] R. F. de Oliveira and C. Büskens, "Benefits of optimal flight planning on noise and emissions abatement at the Frankfurt airport," in *AIAA Guidance, Navigation, and Control Conference*, 2012.

[13] R. F. de Oliveira and C. Büskens, "Emission-optimal flight trajectories with weather hazard avoidance and 4D wind modeling," in *3rd CEAS Air & Space Conference*, 2011.

[14] J. S. Berndt, "JSBSim: An open source flight dynamics model," in *in C++. AIAA*, 2004.