Design of an Architecture for Reconfigurable Real-Time Simulation

Andrew Robbie
Air Operations Division, DSTO
andrew.robbie@dsto.defence.gov.au

Abstract: The requirements of a Human-in-the-Loop simulation software environment for support of research in Human Factors and Human-Computer Interaction are quite different from those for turn-key training simulators. The Air Operations Simulation Centre of DSTO has developed a highly reconfigurable simulation software architecture to satisfy these requirements. The system provides a flexible guaranteed QoS communication framework based on an innovative variant of the publish and subscribe pattern, and a standard interface for running modules with real-time periodic scheduling. The current architecture is assessed in light of recent advances in COTS middleware such as Real-Time CORBA and lower cost PC-based computing hardware.

1. INTRODUCTION

One of the key areas of research at the Air Operations Division of DSTO is Human-in-the-Loop (HiL) Simulation Technology. The Air Operations Simulation Centre (AOSC) acts as a facility to support various research tasks, which include work on Human Factors (HF) and Human Computer Interaction (HCI), distributed simulation efforts (such as the Virtual Air Environment (VAE)) and simulation technology research. The AOSC does not provide a standard training simulator, rather it acts as a test bed to try out new concepts where scientific experiments can be carried out.

Due to the broad scope of the experiments undertaken it is necessary for the various hardware and software assets to act as reusable components. This allows mixing and matching of assets like cockpits (F/A-18, F-111 and BlackHawk, Seahawk and tandem rotary wing) with software such as Electronic Warfare (EW) displays, helmet symbology systems, flight models, ship motion models and the DIS gateway.

Since the goal of many experiments is to collect quantitative data, it is important that the state of components is observable. Also, data recorded from components needs to be correlated both with other software systems and real-world phenomena. For the experimental results to be consistent it is important that there is minimal variation in system performance.

A HiL simulator system must operate in Real Time. That is, in order to mimic the operation of real world systems the simulator must produce outputs to meet real world deadlines. In order to meet these deadlines software components need resources like CPU time and memory; when components communicate with each other there are additional requirements. These performance requirements comprise the Quality of Service (QoS) that is required. These factors place special constraints upon the simulator design.

The general term for software that facilitates communication among components is middleware. The AOSC has developed a software architecture and middleware layer that supports these objectives[1]. A description of this forms the first part of this paper.

Recent development of the distributed object middleware CORBA (the Common Object Request Broker Architecture[2]) includes ways of specifying some of the QoS requirements of Real-Time systems, and eliminating unnecessary overhead when used as a local communications mechanism. The second part of this paper examines how CORBA technology might be applied in future AOSC systems.

2. REQUIREMENTS

Overview The general computational method for real-time simulation is to use time-stepped models for each subsystem (rather than a sequence of events). Each model iterates at a certain rate, depending upon how often it must process new input data and the required response time for its outputs. For example, a common goal is to have Image Generation software update the visual scene at a rate of 60 Hz, so as to produce a sense of fluid motion to the viewer. This general simulator design motivates many of the system requirements.

The system requirements have been broken down into several key categories.

Ease of integration Much of the specialised software used comes from external sources or predates the connection infrastructure. It should be easy to integrate this code using a simple wrapper (i.e. using the Façade pattern [3]).

Reusability It must be possible for a component developed for one simulation to be used in a new project. Rework must not be required if the functionality is the same. For this to occur without rework, each module must be decoupled from other modules (that is, only directly coupled to the middleware layer).

Configuration flexibility The collection of components, their connection topology and individual configuration must be stored independently from the modules. It should not be necessary to re-compile a module to use it in a different configuration; similarly, changing which
components talk to each other should not require re-compilation or changing the component's init-time or run-time configuration.

**Reproduceability & Observability** Given the same connection topology, configuration parameters and inputs, a simulation will produce the same results (to a certain precision). All the information required to achieve reproducible results must be available to be recorded, as must the results of various interim computations. Usually it is sufficient to capture all inputs and outputs for each iteration of the model.

**Performance** HiL simulators seek to mimic in real time the behaviour of real aircraft. Thus response to inputs should neither be too slow nor too fast. To ensure that components can communicate quickly with other models, the system should be able to support communication of models running at 120+ Hz; models may carry out iteration at a faster rate internally. The message transport time should be small in comparison to the model iteration time.

**Robustness & Fault Tolerance** Where possible, the data output from a component should automatically be checked for domain consistency. For example, automatic range checking on numeric data.

The failure or performance/accuracy degradation of any one component should not result in system failure. If a component fails it should be possible to re-start it and have it re-join the simulation.

**Distributed components** In order for the system to be easily scalable, and capable of utilizing different hardware components, it is necessary for the middleware layer to support communication between components distributed across several machines in cluster (i.e., with a high speed, low latency interconnect, rather than across a WAN).

### 3. AOSC ARCHITECTURE

#### 3.1. Overview

The AOSC implementation is based upon each machine in a cluster of computers running a message routing service, referred to as the Memory Manager (MM) (for historical reasons). The MM is directly connected to components running on its node via shared memory — these components are called User Models (UM). The MM acts as a Mediator[3] between the UMs, abstracting interface details, enabling location transparency, and providing data replication and caching services. The MM on each node in the cluster communicate via a high speed, low latency interconnect called Reflective Memory [4]. The general hardware topology of the system is shown in Figure 1.

The MM processes also control the scheduling and resource allocation for all modules. To do this they refer to a global configuration file (the *Global Map*) specific to the simulation.

The inputs and outputs of the modules are communicated using a publish and subscribe mechanism[3], in this case split into two phases. At initialisation time the UM publishes which variables it can export. The MM determines, by consulting the *Global Map* file, which variables from other modules the current module will be subscribing to. Further detail on the system is given below.

#### 3.2 User Models

The general structure of most simulator modules is an initialisation phase, an iterating phase, and a termination phase. To integrate a model we provide a standard interface, referred to as the *UM Shell*. The model developer merely provides definition for the functions `um_initP()`, `um_modelP()` and `um_exitP()`, then passes control to our main loop function, `um_mainP()` (see Figure 2). The `um_modelP()` function is called every frame, and this is where all real-time computation takes place.

All publish and subscribe actions are done in the `um_initP()` phase, along with reading configuration files and any other setup tasks. Any tasks which can not be completed reliably in a small multiple of the frame period must be separated from the synchronous operation of the UM. This is discussed further in Section 3.4 — Scheduling.

The publish and subscribe interface is activated using multiple `mm_registerP()` calls (Listing 1). This function registers which variables should be transferred in and out of the module. This interface transfers only basic data types or arrays of basic data types.

By constraining the expressiveness of the interface several benefits are realised:

- Decouples subscribers, which do not have to know about complex data structures defined by other components.
- Makes the data paths more easily observed by third-party modules like the *data logger*. 

---
int um_main(int argc, char * argv)

void um_init(float rate, const char * config_file)
int um_model(void)
int um_exit(void)

int mm_register(const char * variable_name, aosc_io_type_t io_type, aosc_var_type var_type, size_t count, float rate, void * data, const char * description)

/* Example usage */
static float heading, pitch, roll
static int wow // weight_on_wheels

static double gear_hat[3] // Landing gear position

Listing 1: Interface function prototypes

- Quicker to marshal and un-marshal inter-module messages.
- Allows better real-time bounds on the time taken to send and receive data.
- Because it is such a simple format it is easier to create and use multiple language bindings.

The disadvantages include:
- Modules that are closely coupled, or that exchange complex data, must decompose their messages into more cumbersome formats.
- Harder to exchange variable length data sequences.
- If the underlying model uses Object-Oriented (OO) data structures the extra marshalling must be done by hand, which can be error prone.
- Lack of type checking (it is possible to register an input or output as having a different type than the variable being used).

The UM only receives updates at the beginning of um_model), and updates are only sent to the MM at the end of an iteration. This greatly reduces the debugging effort otherwise typical of systems with many asynchronous components operating in parallel and facilitates the aims of reproduceability and observability.

3.3. Interconnection

When User Model components register a set of input and output variables they use names like ‘heading’, ‘pitch’ and ‘roll’. However, these are not canonical names or meanings which can be used to automatically connect input/output pairs; also, there may be multiple instances of a component. So, the inputs need to be it mapped to the outputs. This information is stored in the Global Map file. For example, suppose we need to connect the outputs of an F/A-18 flight model instance, called f18_dome, to its IG, ig_dome, we might have what is shown in Listing 2.

The data type of both the input and the output in this case is of type AOSC_FLOAT; however, the unmarshalling code
can automatically convert between compatible types, or display warnings if they are incompatible.

Notice that there is also a mapping from f18_dome to the data logger. If the F/A-18 program were sending a specialised struct to the IG it would be much more tedious and error-prone to record this message, as the data logger would have to know about that data structure.

The process of creating mappings has been partially automated with the `connect` tool. This simplifies the addition of components, which are loaded from the component repository (Figure 3), and allows modules to be connected with a 'wiring tool'. The publish/subscribe pairs can then be examined or modified by clicking on the inter-component links (Figure 4).

`Connect` is able to examine components from the repository to determine their inputs and outputs; it does this by querying the binary image of the component, as if these were in an associated IDL (Interface Definition Language) file.

```
NAME_MAP f18_dome_to_ig_dome
VAR_MAP ac_heading ep_phi
VAR_MAP ac_pitch   ep_theta
VAR_MAP ac_roll    ep_psi

NAME_MAP f18_dome_to_data_logger
VAR_MAP ac_heading var_floatsU0
VAR_MAP ac_pitch   var_floatsU1
VAR_MAP ac_roll    var_floatsU2
```

Listing 2: Example of Global Map connection syntax

Figure 3: Connect module view

### 3.4. Scheduling

In order to specify and obtain the required QoS, a scheduling framework is needed to organise the operation of the system. There are many different scheduling techniques [5][6]; these can be separated into static (in which task priorities do not change according to demand) and dynamic (where they do).

The AOSC system uses static periodic scheduling for all tasks which interact with the MM. Every task (a task being the UM thread executing `um_model()`) is allocated a period of time at regular intervals and all UM tasks are considered to have the same priority level.

The scheduler works by dividing time into frames, each of which is 1/60th of a second (though other rates can be chosen if convenient). Modules have execution periods which are integer multiples of the basic frame period; this is more conveniently expressed as a frequency (ie 1x/60Hz, 2x/30Hz, 3x/20Hz).

If multiple tasks have been allocated to the same CPU they operate sequentially. This is done to reduce context switching and other operating system overhead.

Tasks are not restricted in how much of the frame period quantum they can use, and are in fact co-operatively multi-tasked with other tasks on their CPU. This is due to the difficulty of estimating the computation time in complex modules. All modules must complete in one frame in order for the system to be schedulable, so preemptive scheduling would not achieve significant benefits.

To help achieve the aim of reproducability it is desirable that the main system components are synchronised. By this we mean that all modules begin iteration at the same time (hence with the same view of all subscribed variables) and proceed in lock step, that is, after n frames of period p, all modules iterating at rate 1/p will have completed n iterations.

All modules wait on a barrier before beginning iteration. This barrier is toggled at the start of frame to start modules running. There is (conceptually) a different barrier for each scheduling frequency (eg barrier_60Hz, barrier_30Hz, etc) and these are also synchronised, i.e. the 30Hz barrier is released on even multiples of the 60Hz barrier. The timing diagram in Figure 5 shows a snapshot of the scheduling system.
Due to shortcomings in operating system capabilities, tasks allocated to overloaded CPUs are not dynamically migrated to currently idle ones when the system is running in the isolated and scheduling disabled mode. For this reason task allocation to a CPU is presently done manually using experimentation and experience, which in any case has proved acceptable.

The communications channel between the MM and attached UMs utilises shared memory to provide an asynchronous communications service with passive receive. By passive receive we mean that the task receiving the message has to do no work (and hence need not be scheduled). Hence, the MM can receive updates from UMs as they complete without needing to be active. Similarly, when the MM sends updates to UMs they do not need to be woken to service the update message, only when the frame sync barrier is released. This allows the MM to be single threaded, which results in a much simpler design.

Using the periodic method of scheduling allows for simple calculation of transport delays associated with calculating any particular output, as all that is required is adding up the frame delays of the modules which are on the critical path to the output. The corollary of this is that jitter in the transport delay caused by internal processing time variation is small — most jitter arises from lack of synchronisation between external inputs and the simulator, though this can be dealt with. For example, pilot head motion tracking is used for tasks involving HMD symbology, simulated HMD Night Vision systems and three dimensional audio cues. To minimise the delay experienced by processing systems in receiving the latest head position estimates, we need an estimate to arrive from the tracker just as the system is ready to process the next input. The MM or a UM is capable of sending a synchronisation pulse which can be used by appropriate tracking hardware[7] to deliver updates at the correct time.

![Figure 5: Scheduling of UM tasks](image)

### 3.5. Ensuring QoS

To ensure that each component receives the required QoS we have used a resource pre-allocation policy throughout the design. Although this defeats the resource sharing mechanisms of the host OS, we have found that general purpose operating system features are inadequate for meeting the required performance levels. At present no automated admission control system is used.

Each task is allocated to a CPU of a specific cluster node and locked to that CPU (from which other unauthorised tasks are barred). OS scheduling and signals are then disabled for that CPU. In addition, the process address space is locked into memory to prevent paging during operation. Memory allocation (eg using malloc()), socket opening and similar resource acquisition methods are best confined to the initialisation phase (um_init() in UMs).

The communication channel with the MM is via shared memory. Simple locking mechanisms are used in favour of messages queues [8] as these involve more overhead and can result in undesired OS interaction.

Asynchronous processes communicate with synchronous ones via shared memory; care must be taken that the asynchronous process does not hold access to the critical region for significant periods of time. Certain asynchronous processes make use of custom non-blocking multi-thread safe queues to exchange data.

The Memory Managers exchange updates across a low latency Reflective Memory (RM) interface. This allows a range of memory to be mapped from the card address space directly into process memory without intervening OS layers. This area of memory is ‘shared’ between all nodes on the network, with a change on one node automatically propagated to other nodes. The Master MM allocates regions of this address space to the other MMs involved. The RM also allows the system to use a global clock pulse, generated by the Master MM, to synchronise all nodes. This greatly reduces jitter in module update timing.

Most of the functionality to achieve these tasks is confined to custom IPC (Inter Process Communication) libraries, which provide very simple interfaces for UM developers and hide the complexities of dealing with shared memory, locking and threads on several platforms. For example, shared memory might be obtained using Unix System V mechanisms, SGI Arenas or POSIX arena variants of mmap() shared memory.

### 3.6. Configuration

In order to start a simulation component it needs to be given parameters for the UM and resource allocation information. This information is read from the first part of a Global Map file (Listing 3), and comprises:

- **Instance**: the name of the instantiated component.
- **Executable**: the name of the executable.
• Host: the node which will host the component.
• Display: the X-Windows display to use (optional).
• Rate: the scheduling frequency of the module.
• Asynchronous shared memory key ID (optional).
• Configuration file name.

<table>
<thead>
<tr>
<th>Process</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f18_red</td>
<td>ghar</td>
</tr>
<tr>
<td>f18</td>
<td></td>
</tr>
<tr>
<td>p1_blue</td>
<td>elric</td>
</tr>
<tr>
<td>p2c</td>
<td></td>
</tr>
</tbody>
</table>

Listing 3: Global Map module setup information

These parameters are used by the mapstarter program of initiate a simulation. This is a slight variant of the service configurator pattern[9].

4. FUTURE DIRECTIONS

The AOSC simulation infrastructure is continually evolving. Two areas for future research in the area of system architecture will be explored: the possible use of Real-Time CORBA and how the new capabilities offered by low cost PC clusters might shape architectural design.

4.1. CORBA

What is CORBA? The Common ORB (Object Request Broker) Architecture, or CORBA, is a middleware design managed by the Object Management Group (OMG). CORBA is designed to support distributed computing across a range of different hardware, operating systems and languages. An excellent introduction to CORBA is given in Henning and Vinoski [10].

If the AOSC were re-implemented using CORBA technologies we would need to maintain the same level of real-time performance, in addition to obtaining the benefits inherent in CORBA, such as support for OO data structures, type safety, etc. However, previously CORBA has suffered from poor real-time performance. The most obvious way to use CORBA would be with the Event Service — this has a similar structure to the AOSC MM/UM architecture, but lacks real-time capabilities. The TAO ORB, developed by the Distributed Object Computing group of Washington University, is an Open Source ORB designed for high performance. O’Ryan et al. report[11] on the development of an architecture for distributed simulation using real-time extensions to the CORBA Event Service. A real-time Event Service, in combination with other real-time additions to the standard, could provide the basis for future simulation system infrastructure.

Real-Time CORBA A relatively recent addition to the CORBA standard, Real-Time CORBA (RT-C) provides interfaces for specifying the QoS that clients and servers must provide.

The communication QoS aspect is mainly concerned with transport level reliability and resource allocation strategies, rather than mandating quantitative measures of the communications channel like message delay, loss rates or admission control.

For example, when activating a method on a remote object, normal CORBA leaves it to the ORB to decide the connection management issues. For IIOP (Internet InterORB Protocol), the ORB has the choice of using:

1. Unreliable UDP: lowest latency, least reliability, minimal memory consumption.
2. One-time TCP: reliable, poor latency and only temporary memory consumption.
3. Semi-permanent TCP: reliable, moderate to good latency depending on various factors, and continuous memory usage on client and server.

RT-C allows the client (through its RTContext) and the server object (via its IOR (Interoperable Object Reference, similar to a pointer)) to specify the desired strategy. For example, method 1 might be best for sending non-critical messages to systems with small amounts of memory; method 2 may be best if the client is contacting many hosts (ie hundreds) but with little temporal locality, and reliability is more important than initial delay; and method 3 used for systems which desire pre-allocation of communications resources. Possible elaborations include having separate channels for messages of different priorities or requesting a non-multiplexed channel (to reduce queuing delay), or ensuring connection setup is complete.

Giving method invocation messages a priority value allows the underlying transport’s QoS features to be automatically utilised. For example, the IP (Internet Protocol) packets used by IIOP have a header field for Precedence and Type of Service (TOS). Routers can be configured[12] to prioritise traffic using the Precedence field and use the TOS field to make more informed routing decisions to satisfy goals, such as choosing the most reliable route or the route with least delay. Transports such as ATM are even more suited to dealing with RT QoS requirements, but require a more coordinated approach than IP.

Another aspect of QoS configuration arises when more than one possible transport method is available; for example, Unix domain sockets (AF_UNIX) are faster for intra-node communication than TCP/IP (AF_INET), and should be used in preference if possible. Thus we require location transparency but not ignorance. RT-C also deals with how prioritised requests should be serviced. For example, how many threads should be used for servicing events of various priorities and how many threads should be pre-allocated.

A number of issues, e.g., with how RT-C priorities should be mapped onto local OS and transport priorities, are not addressed in v1.0 of RT-C, but should be defined in v2.0.

Pluggable Protocols One of the more promising recent CORBA implementation developments has been the idea
of pluggable protocols. This allows adding new ways for components to communicate by developing a GIOP (General Inter-ORB Protocol) compatible interface to a transport mechanism and loading into the ORB. For example, a GIOP interface which uses System V shared memory (SHMIOP, or SHared Memory Inter-ORB Protocol)[13] has been developed for TAO.

Further development of SHMIOP (hopefully using the POSIX shared memory system) should greatly improve the applicability of CORBA to real time simulation systems by removing the need for all transactions to go via an OS protocol stack (i.e. the TCP/IP stack, which has been demonstrated to be a significant performance bottleneck). User-space communication interfaces can overcome these problems[14]. A pluggable protocol for low-latency interconnect hardware is being investigated, for use with Myrinet[15] or GigaNet[16] (which uses the VIA[17] interface).

**HLA interaction** RT-CORBA is particularly suitable for use in implementing HLA[18]. In fact, the DMSO RTI-NG v1.3 (Run Time Infrastructure) uses the TAO ORB. Having a simulation infrastructure capable of dealing with objects directly, rather than via logic in the HLA gateway, may increase flexibility.

### 4.2 Low Cost Clusters

A quick analysis of the economics of standard PC hardware, compared to the small supercomputer class hardware often used in flight simulators, shows that transferring work to PC systems is desirable. In general the simulation community has mainly been captivated by the PC-IG aspects; however the general computation application aspects also show promise. Groups interested in batch-mode number crunching (e.g., those interested in astrophysics and fluid dynamics) have led the development of low cost clusters of basic PC hardware, running the free Linux OS and communicating with Fast Ethernet; this is generally referred to as a Beowulf system[19]. This type of system can provide excellent price/performance ratios and can easily be expanded and reconfigured.

Using this technology for Real-Time HiL flight simulation experiments presents several difficulties. One concern is that communication between nodes of the cluster takes too long. There have been a number of efforts in dealing with this; the best solution seems to be buying special low latency interconnection hardware like Myrinet, as mentioned in the previous section. However, these devices only provide a fast transport mechanism.

A software abstraction layer is required to enable fast and flexible development. PVM (Parallel Virtual Machine)[20] provides a simple environment but makes too many performance compromises. MPI (Message Passing Interface)[21] does not provide the same ease of use, but is often faster; the MPI-RT (Real-Time) initiative[22] is also promising. Neither of these methods provide much support for OO data structures, and are mainly designed for numerical computation. The potential and difficulties of CORBA have already been mentioned.

The general system topology would be as shown in Figure 6. ‘Mediator’ is the new term for the Memory Manager functionality.

---

**REFERENCES**


**GLOSSARY**

| AF_INET | Address Family — Internet |
| AF_UNIX | Address Family — Unix |
| AOSC | Air Operations Simulation Centre |
| ATM | Asynchronous Transfer Mode |
| CORBA | Common Object Request Broker Architecture |
| DIS | Distributed Interactive Simulation |
| DMSO | US Defense Modelling and Simulation Office |
| DSTO | Defence Science & Technology Organisation |
| GIOP | General Inter-ORB Protocol |
| IPC | Inter-Process Communication |
| HI | Human-in-the-Loop |
| HLA | High Level Architecture |
| HMD | Helmet Mounted Display |
| IDL | Interface Definition Language |
| IG | Image Generation |
| IO | Internet Inter-ORB Protocol |
| IOR | Interoperable Object Reference |
| IP | Internet Protocol |
| MM | Memory Manager |
| MPI | Message Passing Interface |
| OMG | Object Management Group |
| OO | Object-Oriented |
| ORB | Object Request Broker |
| OS | Operating System |
| PC | Personal Computer |
| PVM | Parallel Virtual Machine |
| QoS | Quality of Service |
| RM | Reflective Memory |
| RT-C | Real Time CORBA |
| SHMIOP | SHared Memory Inter-ORB Protocol |
| TCP | Transmission Control Protocol |
| TOS | Type of Service |
| UDP | User Datagram Protocol |
| UM | User Model |
| VAE | Virtual Air Environment |
| VIA | Virtual Interface Architecture |
| WAN | Wide Area Network |