FRAMEWORK IMPLEMENTATION

This part of the dissertation defines the requirements for the software framework based on existing experience within the Command and Control domain. The focus quickly shifts to the framework design and implementation. The framework implementation is then evaluated and the evaluation covers enough aspects of the implementation to address how well the design fits the original requirements. Critical components of the software framework design are formally described and evaluated. This is done using the software architecture concepts and behaviour analysis techniques reviewed in the previous part of this dissertation.

7. Framework Requirements

This chapter marks the start of the third part of this dissertation, which discusses the design and implementation of the proposed simulation software framework. The *use case* for the software framework is created based on existing experience with M&S as well as the requirements, identified in the previous part of this dissertation, for supporting the C2 Enterprise. The *use case* is then used to generate the requirements for the framework implementation.

7.1 Framework Use Case

Figure 7.1 shows a simple *use case* diagram for the software framework. There are five different actors defined for the framework: the software developer, the operator, the C2 system, the LVC simulator(s) and the user or stakeholder. The developer has to be able to extend the framework and use it to develop applications according to the user's requirements. The operator has to be able to use the developer's tools and applications within the C2 enterprise. The user may also be the operator.

The use case shows that the developer can also be the operator, but this is mainly for testing and evaluation. The operator is responsible for running the application and possibly managing the simulation execution or the links to external systems. The operator could also form part of the simulation in the case of virtual simulations or human-in-the-loop (HIL) type applications. The operator might require additional feedback or insight through a debriefing capability provided by the application. For this the application should be able to capture the relevant information and present it in a sensible way.

Developers have several tasks: they have to maintain the framework and ensure that the code-base is of a high enough quality; they have to build and test the application(s); they have to extend the framework to accommodate new features; and, they have to build the relevant models of systems in order to virtualise them.

The external C2 systems and LVC simulators can be anything from real equipment like search radars, aircraft and ships, to simulators used for training and/or planning.

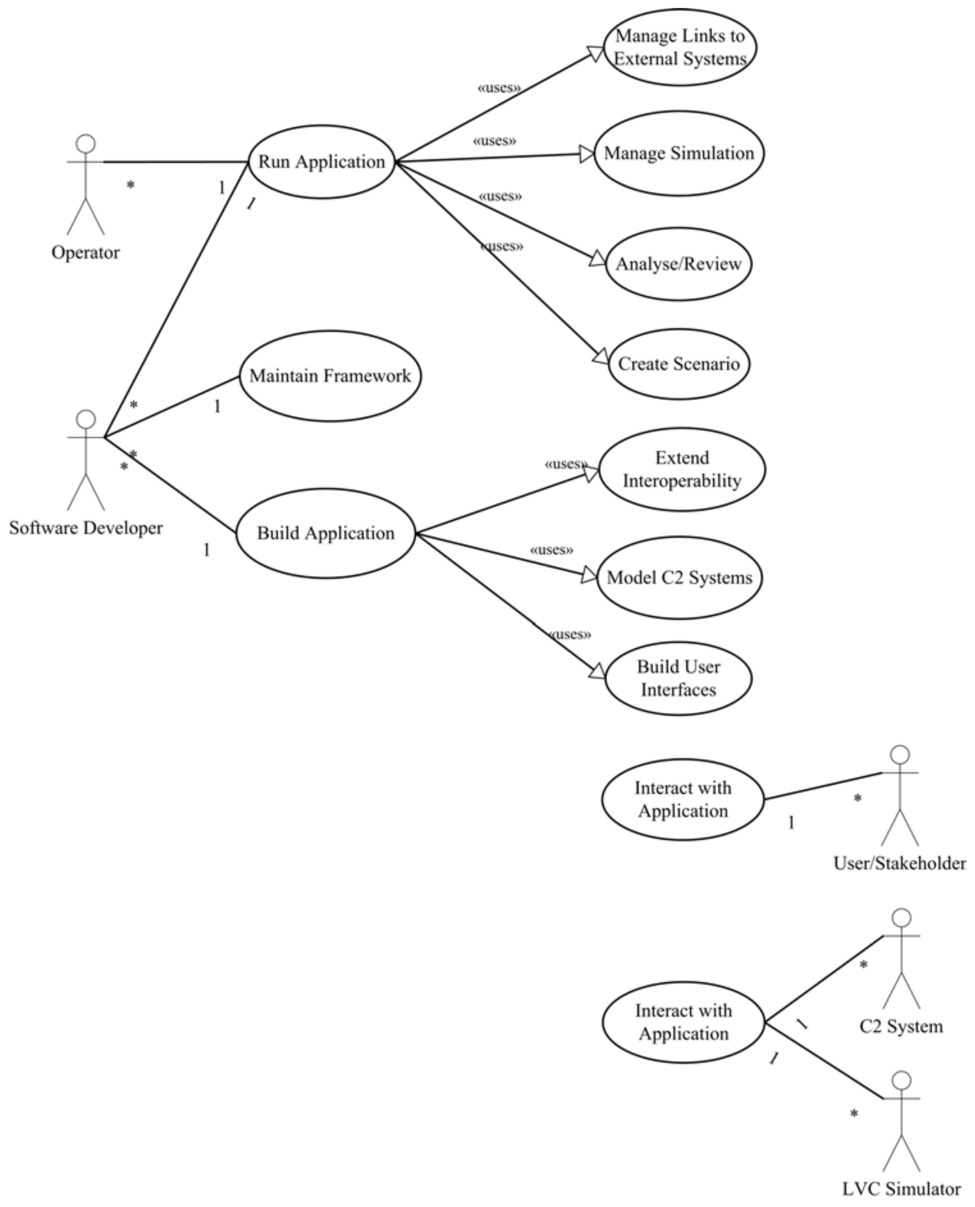


Figure 7.1: Framework Use Case Diagram

7.2 Framework Requirements

The software framework requirements are generated from the *use case* diagram. The focus is placed on supporting the tactical networks that feed data into and out of the C2 enterprise. The interoperability, M&S and application development capabilities provided by the framework supports the enterprise by enabling the integration of the tactical layer with the enterprise business layer and by filling any gaps in the tactical layer.

The knowledge gained during the literature review, presented in the previous part of this dissertation, also contributed to formulating the framework's requirements. These requirements can be divided into five main points: interoperability with C2 systems, virtualisation of C2 equipment using M&S, application development, good code quality and performance and portability. Figure 6.2 in Chapter 6 shows how the framework capabilities should fill the gap between the C2 Enterprise and the operational tactical systems.

7.2.1 Interoperability with C2 Systems

To function within the C2 enterprise the framework should enable the following:

- interoperability with legacy and net-centric C2 systems and simulators,
- protocol translation when communicating with systems and simulators,
- automatic attribute translation when translating to and from external data representations,
- unified and extendable internal information model,
- protocol bridging (acting as an adaptor or gateway for systems that do not support the correct protocol or interface).

7.2.2 Virtualisation of C2 equipment using M&S

Applications and tools created with the framework can support the C2 enterprise by deploying virtual systems when the real systems cannot be deployed. This introduces the following requirements:

- dynamic addition and removal of simulation objects like services and models,
- operator in the loop (OIL) support,
- running in real-time and the ability to catch up if the simulation was slowed down temporarily (soft real-time),
- running in reverse, running as fast as possible and pausing execution,
- the ability to jump in time, and
- a configurable frame rate.

It should be possible to distribute the execution over multiple nodes for increased performance and fault tolerance.

7.2.3 Application Development

The users and operators require a means of interacting with the virtual environment or controlling the system interoperability. This interaction could be through anything from a text console to a full Graphical User Interface (GUI). The framework should provide the software developer with a concise way of integrating with user interfaces and applications. The framework should also define how the application should interact with the underlying virtualisation and interoperability capabilities of the framework.

7.2.4 Good Code Quality

The framework is intended for rapid development of technology demonstrators and prototyping of software. More often than not the applications are also subject to ad-hoc changes in user requirements. The quality of the framework code base will determine how the framework is used. The framework code base should be characterised by the following:

- a common interoperability infrastructure across the tactical environment,
- a common M&S infrastructure across the tactical environment,
- seamless integration of the M&S and interoperability infrastructures.
- unified data collection and analysis,
- fault-tolerance and reliability,
- usability, maintainability and extensibility

7.2.5 Performance and Portability

Real-time performance is desired, but in most cases this is actually *soft* real-time since the targeted operating systems do not support *hard* real-time. Parallel execution (distribution over multiple CPU/Cores on one node) should also be considered to utilise the power of the new generation of multi-core processors. There are cases where interoperability is required with equipment and systems that might have very strict timing requirements: this requires that some components of the framework have to be run in separate high priority threads to achieve the desired execution speed and reliability.

Ultimately the framework should make it easy for application developers to create good quality applications and tools that support the C2 Enterprise. The framework design and implementation should not be specific to any platform or operating system. The next chapter discusses the current framework design and implementation.

8. Framework Design and Implementation

This chapter discusses the design and implementation of the software application framework. The framework architecture is also described using the various software architecture concepts and terms reviewed in part two of this dissertation.

8.1 Design Overview

The framework is split into five functional layers: the *backbone* layer, the *infrastructure* layer, the *interoperability* layer, the *simulation* layer and the *application* layer. Figure 8.1 shows the layers of the framework mapped onto specific layers of the OSI model to give some perspective on the functionality of each layer.

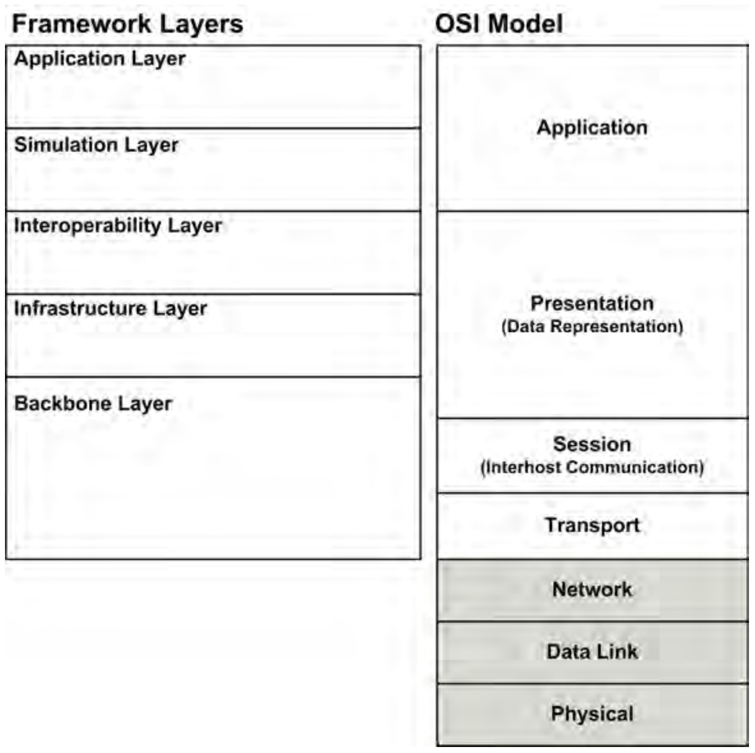


Figure 8.1: The Framework Layers

The layers are implemented as separate C++ libraries. There is a clear separation between the layers in terms of functionality and each layer can be compiled or modified without affecting the other layers. This layered architecture accommodates multiple teams of software developers either working on different layers of the framework or building different applications using the framework.

The work discussed in this dissertation covers all the layers depicted in Figure 8.1, but the framework implementation focuses on the *backbone*, *infrastructure* and *interoperability* layers. The *simulation* and *application* layers are more application specific, with some examples given in the next chapter. This chapter does however discuss the *simulation* and *application* layers in enough detail to show how these layers should be extended and how the framework interfaces with higher level user applications.

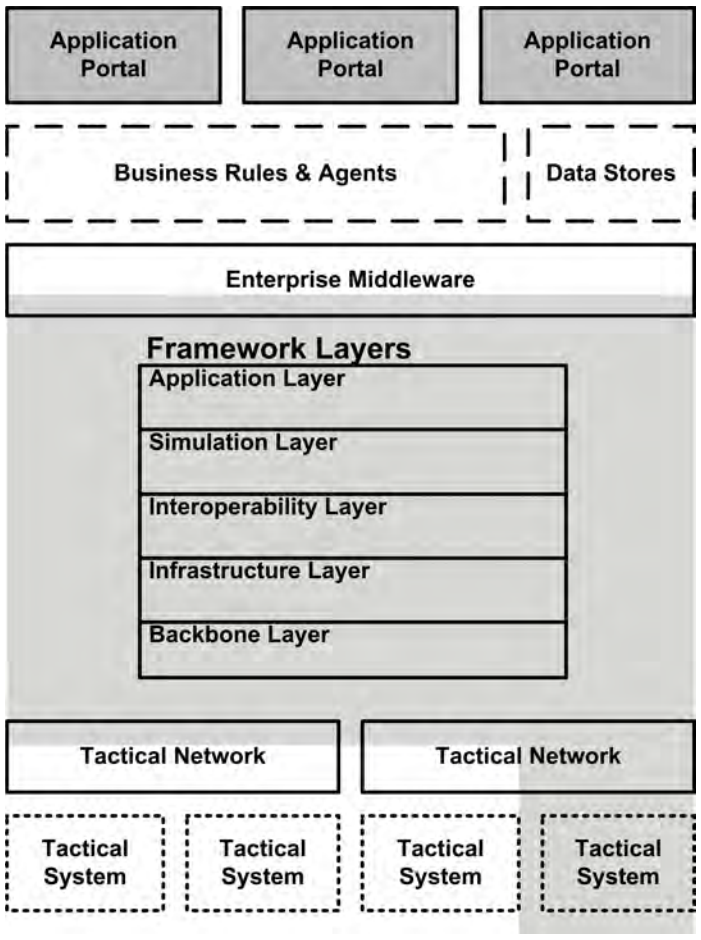


Figure 8.2: The Framework Supporting the C2 Enterprise

Figure 6.2 in Chapter 6 shows that the C2 enterprise consists of a tactical network layer and a enterprise layer—two distinct layers. Figure 8.2 shows how the framework supports the integration of the tactical layer with the enterprise business layer (the framework enables *interoperability with C2 systems*). The framework also helps fill any gaps in the tactical layer through *virtualisation of C2/tactical equipment using M&S*.

The use of proven cross-platform libraries improves the quality and usability of the applications created with the framework. The framework and current application implementations make use of the following open-source or standardised libraries:

- The framework relies extensively on the *Standard Template Libraries* (STL) for C++ for data structures, containers, sorting, etcetera. STL is proven, well documented and comes standard with C++.
- The framework also relies on some of the *Boost* C++ libraries (*http://www.boost.org*) for things like multi-threading, string hashing and random number generation. *Boost* is a free cross-platform set of libraries with some of the libraries being considered for inclusion into the new C++ standard (currently being finalised).
- The applications created with the framework make use of user interface technologies like wxWidgets (*http://www.wxwindows.org*) and Qt (*http://qt.nokia.com/products*) for building the relevant user interfaces. *wxWidgets* and *Qt* are cross-platform and open-source—either can be used to create good user interfaces for applications.
- The applications use *Open Scene Graph* (OSG, *http://www.openscenegraph.org*) to create 2D and 3D visualisation panels in the user interfaces. OSG is a cross-platform, open-source 3D graphics toolkit.
- The framework builds on *libxml* (http://xmlsoft.org) to create a simple object-oriented XML reading and writing capability. *libxml* is a cross-platform XML processing library for C.

The remainder of this chapter discusses the five layers of the framework.

8.2 The Backbone Layer

The *backbone* layer provides Inter Process Communication (IPC) and object execution (i.e. functionally it corresponds to the OSI Transport and Session layers, as well as in part to the Presentation layer, as shown in Figure 8.1). The backbone also contains a set of core components that contribute to the portability and quality of the framework.

The executable objects are commonly referred to as *backbone objects.* The backbone object execution can be distributed among multiple hosts or among multiple CPUs of a single host (or both) (see Figure 8.3). The process responsible for managing a set of backbone objects is referred to as a *backbone node.* Data flows from one backbone object to another in the form of *issues*, where issues encapsulate events which are called *titles.*

The backbone is loosely based on the work described in (Duvenhage and Kourie, 2008): objects communicate with each other using topic-based publish/subscribe-type message passing (Eugster et al., 2003). The backbone architecture can also be described as having an *event-based, implicit invocation* style, since the backbone objects exchange data through the backbone and do not access each other directly (see Chapter 3). In this context the backbone layer follows the *mediator* design pattern, since backbone objects only communicate through the backbone and not directly with each other (see Chapter 3).

8.2.1 Inter Object Communication

What issues a backbone object can publish and where the issues go are determined by the publications which backbone objects register and by the subscriptions which backbone objects

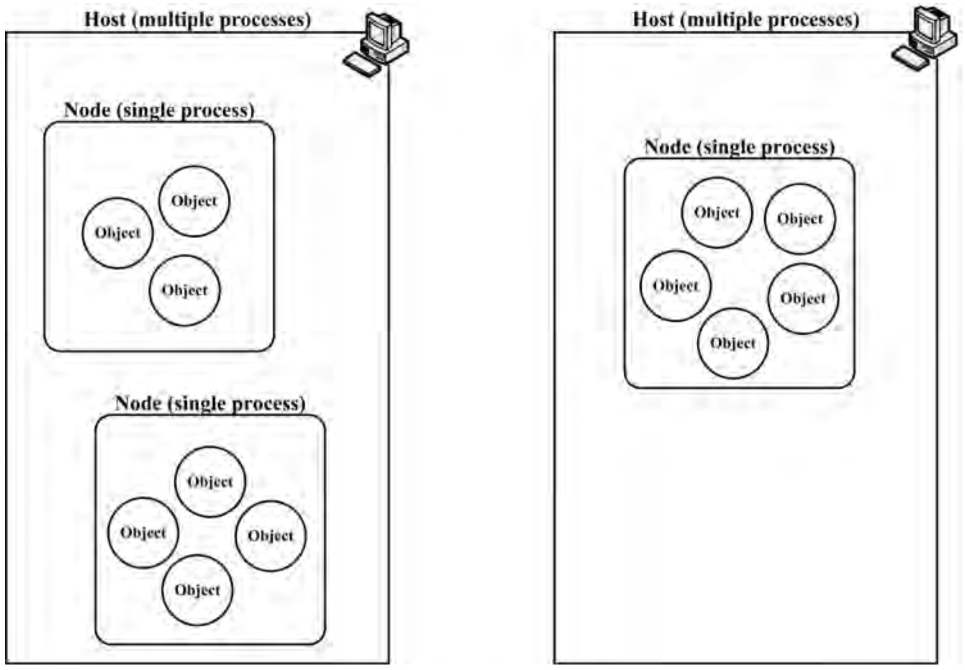


Figure 8.3: The Hosts, Nodes and Objects of the Framework

make. Issues are also inherently generated by the backbone layer for each backbone object in the following cases:

- The backbone generates a publication issue whenever an object registers a publication. The issue is then broadcast to all objects on all nodes.
- The backbone generates a subscription issue whenever an object registers a subscription. The issue is then broadcast to all objects on all nodes. Any object that has a matching publication then processes the subscription.
- The backbone generates a subscription issue in response to the delivery of a publication issue if there is a relevant subscription. The subscription issue is then sent to the publisher.

This passing of issues allows backbone objects to register and delete publications and to add and remove subscriptions to other titles in an *ad-hoc* fashion (i.e. during runtime). Registering a publication will trigger a subscription issue from all objects that have an interest in the publication. Making a subscription will create a subscription issue that is processed by all the objects that have the relevant publication registered.

The title interface contains methods for streaming and de-streaming the title attributes to and from a binary stream. Title objects are automatically identified and constructed (this is explained in the next part of this section) and are converted to and from binary when transmitted from one node to another. This makes it very easy to support any number of title types without having to modify the backbone layer.

8.2.2 Inherent Object Construction

The built in runtime type information of C++ is not good enough to uniquely identify the type of objects. Because of this the backbone includes its own type system, referred to the *object hierarchy*. The backbone layer also contains an *object factory* that can automatically construct any class within the *object hierarchy*.

A class can be added to the *object hierarchy* by inheriting from a specific interface and including the relevant class members These class members are then used to identify classes within the object hierarchy. Classes are identified within the hierarchy in two ways: based on the class name (string value) or based on the hash value of the class name (for faster identification).

The object type information also indicates its parent type which allows an object to be identified by the object's type as well as by the type of any one of the object's parents within the object hierarchy. Operations to check the type of objects as well as perform safe casting between types are available as part of the backbone.

The *object factory* follows the *factory method* design pattern (see Chapter 3) and enables automatic construction of objects based on its type (as defined in the object hierarchy). Titles and backbone objects are part of the object hierarchy. The *object factory* and *object hierarchy* make it possible to identify and construct titles with the correct type when reading data from other nodes—other layers of the framework as well as applications can add new object and title types without having to explicitly specify them in the backbone layer.

8.2.3 Distributed Object Execution

The backbone runs at a fixed frame rate which determines the simulation time step size. The backbone calls an object to give it time to update itself and read and publish issues. Each backbone object has a very simple interface that is called by the backbone. Not all objects would have to run on every frame and for this an object can specify a *trigger-frame* which specifies the intervals at which the object should be called.

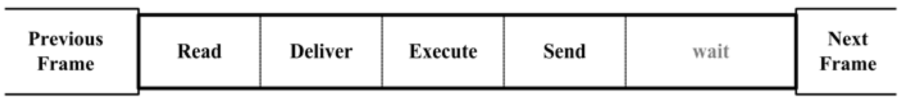


Figure 8.4: The Backbone Object Execution Frame

The backbone executes each simulation frame in five steps with conservative (or lock-step) time management between nodes (see Figure 8.4).

1. The backbone receives all the issues that were published in the previous frame. The backbone keeps on reading issues until all nodes are finished with the previous frame.
2. The backbone then delivers the issues to the correct backbone objects.
3. The backbone then calls all the backbone objects that have a trigger frame matching the current frame. The backbone objects update themselves and get a chance to publish any new issues. Newly published issues are temporarily stored in the backbone.

4. The backbone then sends out all the published issues to the other nodes.
5. The backbone then indicates to other nodes that it is finished with its current frame and continues to the next frame.

The backbone layer uses a separate component, called a *hub*, to transfer issues from one node to another. The hub manages the inter process communication (IPC) without affecting the rest of the backbone layer. The hub interface is part of the backbone layer, but the hub implementations are part of the infrastructure layer and will be discussed in more detail in the next section. The node also uses the hub interface to signal the end of its current frame and then to wait for all the other nodes. The application should then wait for the proper period of time before starting a new frame to keep to the relevant object execution frame rate (see Figure 8.4).

Backbone objects always publish issues for the next frame and the backbone only sends out those issues once all the objects have been called. This can be seen as a form of double buffering, since objects only have access to new issues in the next frame. This situation is ideally suited to parallelisation and the backbone objects can be executed concurrently within a frame. For this the backbone divides the object execution among several worker threads to better utilise the potential of multi-processor systems or multi-core CPUs.

The worker threads execute objects concurrently (and independently). Each worker thread has a fixed set of backbone objects it executes. Each worker thread is also responsible for delivering the relevant issues that were published in the previous frame to its set of backbone objects. The worker threads all use the same set of delivered titles since these titles are not modified by the backbone objects.

There might be multiple hub implementations in the framework and it is assumed that the hub implementations are not thread safe (i.e. allowing multiple threads to access the hub at the same time would result in undefined behaviour or errors)—this is to make it easier to implement new hubs. For this reason, one of the worker threads have to execute its objects and then wait for all the other worker threads before sending all the published issues to the *hub*. This worker thread is also referred to as the main thread.

8.2.4 Subscriptions and Publications

The backbone manages publications and subscriptions as queues of titles. A backbone object can push data onto the back of its publications and the backbone would pop the data off the front of the object's publications when the object execution has finished. The backbone also pushes titles onto the back of an object's subscriptions and an object could then pop the titles off the front of its subscriptions. Each backbone object has direct access to all the publications and subscriptions it has registered. The subscription and publication classes have specific interfaces that allow the backbone object to push titles to publications and pop titles from subscriptions.

The publication and subscription interfaces do not however accommodate quantisation of states or integration of events (see *QDEVS* at the end of Chapter 5). For this the backbone layer includes several template functions that control how titles are pushed to publications or popped from subscriptions. This enables title quantisation at the publisher and title integration at the subscriber, which makes things like *dead-reckoning* possible. The template functions, combined with the object type casting and checking provided by the backbone, control how sets of titles are updated and published and then rebuilt at the subscriber.

8.2.5 Core Backbone Components

The backbone contains a set of core components that contribute to the portability and quality of the framework. The backbone abstracts aspects such as multi-threading, memory management and networking to be operating system independent. This means that the framework can run on any operating system for which the core components have been implemented.

Multi-threading abstractions are implemented in the backbone using one of the *boost* cross-platform libraries. The framework also makes use of *Boost* mutexes and barriers for thread synchronisation. *Scoped locks* are used extensively throughout the framework to make it easier for the developer to manage the locking and unlocking of resources (see Chapter 3). Process-control abstractions also make it possible to control operating system specific things like thread-priority, thread affinity and process-priority in a operating system independent way.

The backbone includes a custom memory manager that helps track down memory leaks. The memory manager is created as a *singleton* (see Chapter 3). All objects in the backbone object hierarchy inherit from a base class that has the memory operators overloaded to store the file and line number of the allocation and to register the allocation. It is then possible to at any time examine the registered memory allocations. Doing this when a application exits provides the location in the source code of memory allocations that were never de-allocated (i.e. potential memory leaks).

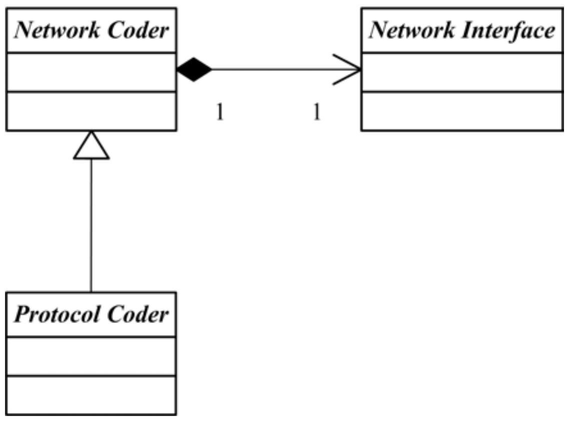


Figure 8.5: The Backbone Network Interface Classes

The backbone provides networking classes that are divided into *network interfaces* and *network coders* (see Figure 8.5). A network interface does the low-level reading and writing of binary data from various interfaces like files, network transport interfaces and even hardware interfaces like RS232. A network coder is a wrapper for a network interface and is responsible for translating the binary data of the interface to titles when receiving data and from titles when transmitting data.

The network interface classes all present the exact same interface, providing a unified way of accessing binary streams (including files). This means that a network coder can operate on any one of the network interfaces, making the networking much more flexible. The network coders also log all the binary data (received and sent) to a file. These *raw* logs can then be used for playback at a later stage by using a file interface with the relevant coder.

The network coders run on separate threads to ensure that interface creation or slow data transfer do not interfere with the backbone object execution. The interoperability layer uses extended network coders referred to as *protocol coders* that operate on various data formats and protocols. Protocol coders are discussed in more detail later in this chapter. The network coder thread also includes mechanisms to recover and possibly re-connect when a network interface fails.

The backbone contains components that can measure the performance of the backbone object execution. This helps to optimise the object execution and distribution. The performance measures look at the following:

- the overall application load, which provides an indication of how well the application in running in general,
- the backbone overhead, which shows how much of the time is spent on modelling versus time spent on reading and writing titles,
- the hub bandwidth throughput, which gives an indication of the utilisation of the underlying transport medium when running in distributed mode, and
- the ratio of titles sent to local objects vs. titles sent to objects on other nodes, which indicates how well the objects are distributed among the different nodes.

The overhead is an indication of the amount of data transported over the backbone and gives an indication of how successfully an application could be distributed. Objects that interact closely, exchanging a lot of data, should typically be located on the same node to minimise inter-node bandwidth usage.

8.3 The Infrastructure Layer

The *infrastructure* layer extends the capability of the backbone layer from simple object execution to the simulation of virtual environments. The infrastructure layer is also responsible for the simulation time management and synchronisation between different nodes.

The backbone objects and basic object titles are extended for modelling and simulation of spatial, time-based phenomena. The backbone objects are also extended to allow saving and loading object attributes in a XML format. The framework uses this to read and write XML scenario files which specify what object are loaded into the backbone.

8.3.1 Spatial Reference and Environment Models

This layer adds spatial reference models for coordinate representation and translation. The models support Meridian and Cartesian coordinates and vectors as well as orientation. This layer also includes the relevant coordinate conversion operations.

The infrastructure layer also adds the environment model for terrain. The environment model can also include things like atmosphere and sun position. The terrain gives objects the ability to get the terrain altitude as well as the ability to test for line of sight. The terrain components are designed to be easily extendable to support different terrain formats. The terrain is loaded as a set of terrain tiles.

8.3.2 The Bootloader

The bootloader introduces the backbone object XML interface. It allows backbone object attributes to be loaded (and saved) from XML scenario files. The bootloader identifies and creates objects from the object hierarchy based on the type of the object. The XML element names correspond to the relevant object class names as defined in the backbone object hierarchy.

Any backbone object that is in the object hierarchy and inherits from the XML interface can be loaded by the bootloader. This, along with the use of the backbone object factory, allows the bootloader to support an arbitrary number of objects without having to modify or recompile the infrastructure layer.

8.3.3 The Node Hub

The backbone uses a separate component, called a hub, to transfer issues from one node to another. The hub implementations are found in the infrastructure layer and not in the backbone layer since it was desirable to be able to configure the hub through the XML scenario file.

The hub specifies the type of inter process communication (IPC) used. It controls the inter-node communication, synchronisation, node addressing and inter-node connection setup. This makes it possible to change the backbone infrastructure from a distributed peer-to-peer TCP scheme to a parallel memory-mapped scheme or even a web-based scheme by using different hub implementations.

The current framework implementation includes two hub implementations. The first is a very simple *single node* hub implementation that just delivers everything that was published in the previous frame to the local node. It is very fast and ideal when all the backbone objects are run on one node (no distribution).

The second hub implementation is a *peer-to-peer TCP/IP node* hub that allows two or more nodes to be connected (i.e. distributed backbone object execution). The hub implementation creates a mesh network with every node connected to all nodes except itself (see Figure 8.6). The hub implementation is then intelligent enough to only send issues to the nodes that have the relevant subscribers. The *peer-to-peer TCP/IP node* hub can be used to run nodes distributed over multiple hosts, but there can also be more than one node per host.

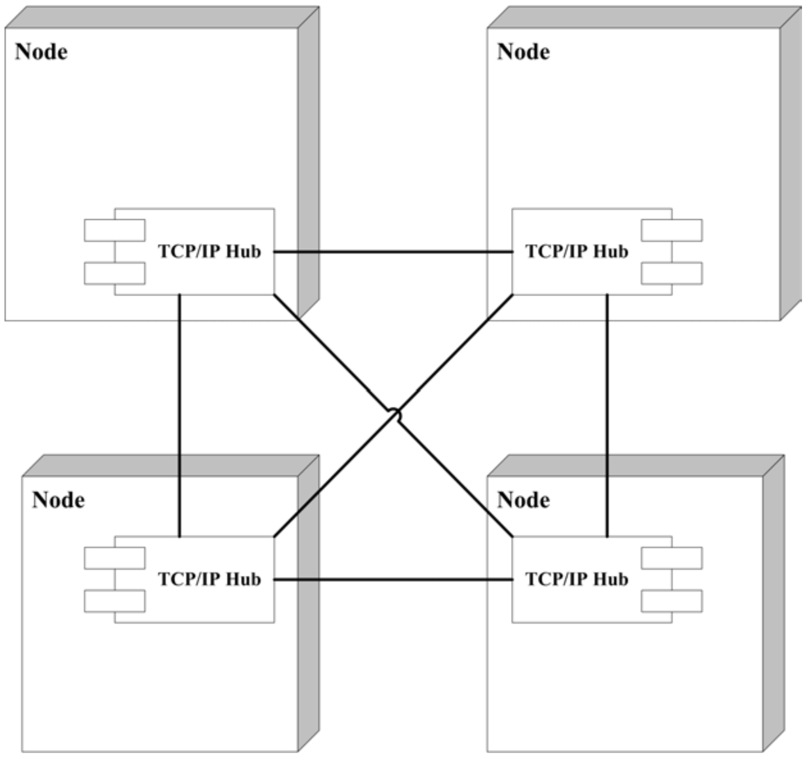


Figure 8.6: The TCP/IP Node Hub Inter-connection

8.3.4 Information Representation and Translation

The data model of a virtual environment implemented with the framework is the specific set of titles published by the simulation objects. A title does however not specify how an object should react when it receives a title through a specific subscription. The data model also includes how, in the military case, objects are classified as units, equipment, weapons, etcetera. For this the current infrastructure layer implements the MIL-STD-2525B warfighting symbology standard.

Real-world systems use tactical data links or proprietary protocols to communicate. The information exchanged by these systems may not match the titles and symbology used by the virtual environment. The internal titles would have to be translated to and from the external data models used by the real-world systems to interoperate with those systems.

Translating titles to and from the various external data models is done by the *protocol coders*, discussed in the next section, but the infrastructure layer contains the components for defining and matching the symbology used by external systems with the internal symbology. For example: the internal data model will identify an fighter aircraft as a *fixed wing military aerial unit* with a *fighter role*; an external system might only have *aircraft* or *bomber* defined.

Tactical data links have very specific ways of addressing different systems in the network, but the backbone uses a single string (the simulation ID) to identify objects. The infrastructure layer also contains components for system address translation, which allow the *protocol coders* to translate to and from external system addresses.

8.4 The Interoperability Layer

The *interoperability* layer adds the required components to interoperate with real-world systems and other simulators. This is where all tactical systems and enterprise services link into the framework. All the relevant protocols and data representations of the legacy and net-centric systems have to be understood and translated. The interoperability layer consists of the following:

- the protocol coders,
- the backbone link objects for the protocol coders, and
- the *gateway* interoperability service.

The *protocol coder* components are extended *network coder* components (see Figure 8.5). Communicating with real-world systems involves creating one or more network coders which are responsible for translating between the *external* systems and the *internal* application components built with the framework. These protocol coders operate on the syntax or structure of the foreign data and only map the information onto titles without understanding the data (i.e. on a syntax level and not on a semantic level).

The *protocol coder* components all run on their own threads to prevent slow or blocking interfaces from holding up the backbone object execution. The coders are responsible for link setup, link tear-down and handshaking (maintaining the link). These functions may be slow depending on the type of interface to the external system and the link requirements. Having the coders run on separate threads makes it very easy to execute these functions without causing the backbone object execution to be delayed—the coder executes independently of the backbone. For example: a coder can wait four seconds for a reply and only then return titles to the backbone with no delay in the object execution.

Normally the coder thread priority is set to *below normal* while the backbone worker thread priority is set to *normal*. This caused the operating system to give the backbone object execution priority above protocol coder execution. This helps to improve the backbone object execution when there are many protocol coders. The drawback of this is that the coder threads might starve (i.e. not be run by the operating system if the backbone object execution is using too much CPU resources).

Some protocol coders have very strict timing requirements and have to run with a very high confidence or at a very high rate. This is typically required when virtualising or emulating systems that normally send messages at very specific intervals. In these cases the developer can set the protocol coder thread priority very high. This commands the operating system to give the coder thread priority above all other threads. The coder execution should however be carefully controlled since high-priority threads can starve the rest of the operating system threads.

The protocol coders are not backbone objects and have to be wrapped inside extended backbone objects, called *link* objects that can be loaded and executed by the framework. The interoperability layer includes a basic *link* object that handles the locking of the coder for thread-safe access; manages the publications and subscriptions; and reports the link-state. The interoperability layer includes many different *link* objects that extend the basic link object.

The basic link also adjusts titles to have the correct time reference: incoming titles are adjusted to be relative to the simulation time while outgoing titles are adjusted to be relative to the clock of the external system. The link buffers incoming titles if the adjusted time reference is in the future. This normally occurs when reading from a file interface (i.e. all the protocol information is available immediately) or when the backbone object execution is too slow (i.e. real-time execution cannot be maintained).

There is also a gateway service which is a backbone object extended to act as a router for the titles from the different links. The gateway service subscribes to titles from all links and selectively publishes titles back to links based on the configured routes.

8.5 The Simulation and Application Layers

The *application* layer (and to a certain extent the *simulation* layer as well) is very application specific. A typical rapid application development code-base would consist of many different simulation and application layer implementations that could be reused or extended to quickly create new tools and applications. The *simulation* layer allows developers to create unique simulations or tools by adding the required models and services.

The virtualisation (or emulation) of systems happens on this level. A simulation model is created by extending a backbone object to simulate the behaviour of an actual system. Models of aircraft, for example, would simulate the flying qualities of a specific aircraft and then publish certain attributes of the aircraft (like position and velocity).

The simulation layer is built on top of the *infrastructure* layer and the *interoperability* layer. The simulation layer is therefore ideally equipped to create models of systems that interface with the C2 enterprise as the real systems would. The next chapter discusses the *Radar Emulator* application, which is an example of this.

The framework will be used to create many different types of applications, but there are some generic components that most applications will share. This is discussed in the next few paragraphs.

8.5.1 Application Integration

Applications might require specific command-line arguments. For this the framework includes, as part of the core operating system abstractions, a command line argument parser that gives any part of an application access to the command line arguments.

Applications also require the ability to load application specific parameters from disk and save them again. The *application* layer provides an extendable XML parameter loader. It works similarly to the *backbone* object boot-loader and application developers can add additional parameter sets.

Applications need an interface for loading and saving scenarios as well as managing the *backbone* object execution. The *application* layer provides a generic *framework execution thread* that manages scenario loading, object execution, object locking and provides access to the spatial environment. This component frees the application from the complexities of loading and saving XML files, locking backbone objects for safe access and managing the object execution. All the applications built with the framework have access to this *framework execution* component.

The *application* layer has to provide a means of collating the capabilities of the framework and integrating it with the relevant technologies for building the user applications and tools. The framework uses *control services* which are normal backbone objects that are instantiated by the user application rather than from the XML scenario file. The user application can create the *control service* and add it to the backbone (using the *framework execution* component) as soon as a scenario is loaded.

Control services can be simple or complex, depending on the application requirements. A control service can subscribe and publish like any other backbone object, but it also allows the application to access the title information through an application specific interface. This means that the application still works with titles, but the control services manage how the information gets processed and then distributed to other backbone objects. Applications can dynamically add and remove control services. One restriction is that an application should lock a control service before it uses it—this is to prevent the backbone from accessing a control service while the application is busy using it.

The next chapter evaluates the framework implementation to determine how well the implementation fits the requirements. This evaluation covers a wide range of criteria. Critical components of the framework are also evaluated using the software architecture concepts as well as the behaviour analysis techniques reviewed in part two of this dissertation.

9. Framework Evaluation

In this chapter the evaluation of the framework implementation is discussed. The evaluation covers enough aspects of the implementation to address how well the implementation fits the original requirements discussed in Chapter 7 (interoperability with C2 systems, virtualisation of C2 equipment using M&S, application development, good code quality and performance and portability).

This chapter is divided into three sections. The first section evaluates the performance and scalability of applications created with the framework. The next section discusses the existing C2 applications created with the framework. The third section formally evaluates the framework implementation. The formal evaluation is done using the software architecture concepts and behaviour analysis techniques reviewed in part two of this dissertation.

This chapter uses the terms *host*, *node*, *object*, *title* and *worker thread* extensively. Please refer back to Chapter 8 for a description of these.

9.1 Performance and Scalability Testing

Object execution performance and scalability is important when virtualising C2 equipment. This section discusses the evaluation of the general M&S performance of the framework implementation. A small test application was created with the framework. It is a very simple simulation that runs multiple instances of the same model. The model has a constant execution time of 2.5ms and only publishes one type of title. The number of titles published are however configurable. The title has several attributes and has a size of 135 bytes when streamed.

The simulation frame rate is set at 100Hz with 160 models in total. Each model subscribes to every other model (i.e. each model will receive 159 *sets* of titles per frame—one set of titles from every model). The model execution is distributed over multiple hosts as well as distributed over multiple CPU cores of one host. The test application is run multiple times with the models evenly distributed over a varying number of hosts and CPU cores. This tests both the distributed performance and parallel performance of the object execution.

The fact that the test application only has one type of model and one type of title makes it easy to analyse the distributed and parallel performance of the test application. Normally one would get several different types of models, publishing different types of tiles at lower rates (each model subscribing and publishing to every other model is actually the worst case scenario).

Figure 9.1: The 10 Toshiba Qosmio Laptops

The hosts used for the tests are Toshiba Qosmio laptops. Each laptop runs MS Windows 7 and has an Intel i7 processor with four hyper-threaded cores. The *hyper-threading* effectively provides eight *execution units* per CPU. Each *execution unit* is also referred to as a *core* in the rest of this chapter (the difference between a hyper-thread and a real CPU core is beyond the scope of this discussion). All the hosts are connected using a 1Gbps Local Area Network (LAN). Ten laptops were used for the tests.

9.1.1 Expected Behaviour

The test hosts can schedule eight threads of execution to run concurrently (one per core). An application running on a single thread only has access to one core at a time which amounts to 12.5% of the total CPU resources. An application that can use eight threads *effectively* should be able to fully utilise the host CPU.

The object execution can be distributed among several *worker threads* in the same node (i.e. parallel execution utilising multiple CPU cores of one host). The objects are divided up equally among the worker threads and the worker threads all operate independently of each other on the same set of delivered titles. Something that may affect execution performance is resource *locking*: the framework uses a memory manager that keeps track of all memory allocations within a node and each node has its own memory manager. Allocating and deleting memory involves locking some parts of the the memory manager. The contention caused by backbone objects allocating or deleting memory from different worker threads may then decrease the execution performance. The contention at the memory manager should be the only thing affecting the worker thread performance.

Distributed model execution involves executing backbone objects on different nodes and transferring data between nodes. The amount of data to transfer increases as more nodes are added, for example: 160 models distributed over two nodes adds up to 80 models per node (i.e. 80 models are publishing to another node, giving 80 sets of titles being received and published per frame by each node); 160 models distributed over four nodes gives 40 models per node (i.e. 40 models are publishing to three nodes, giving 120 sets of titles being received and published per frame by each node). The inter-node data transfer *overhead* in the test application can be controlled by configuring the number of titles each test model publishes per frame. The network throughput is measured as the total amount of bytes sent and received by each node per second.

Nodes can be deployed per host (i.e. distributed) or as multiple processes on the same host. Communication between the nodes occur in exactly the same way in both cases (using TCP/IP). In the first case (the distributed case) the inter-host data transfer and synchronisation may reduce the overall performance and in the second case the overhead should be very low. In both cases the network overhead and host synchronisation should be the only thing affecting the execution performance if there is only one worker thread per node (since resource locking or worker thread contention within the nodes do not affect performance in these cases).

9.1.2 Parallel Performance

The framework object execution can be distributed among several worker threads in the same node (parallel execution utilising multiple CPU cores of one host). The test application was run with varying worker thread counts and detailed results are shown in Figure 9.2. A number of summarised views of this data are presented in subsequent figures.

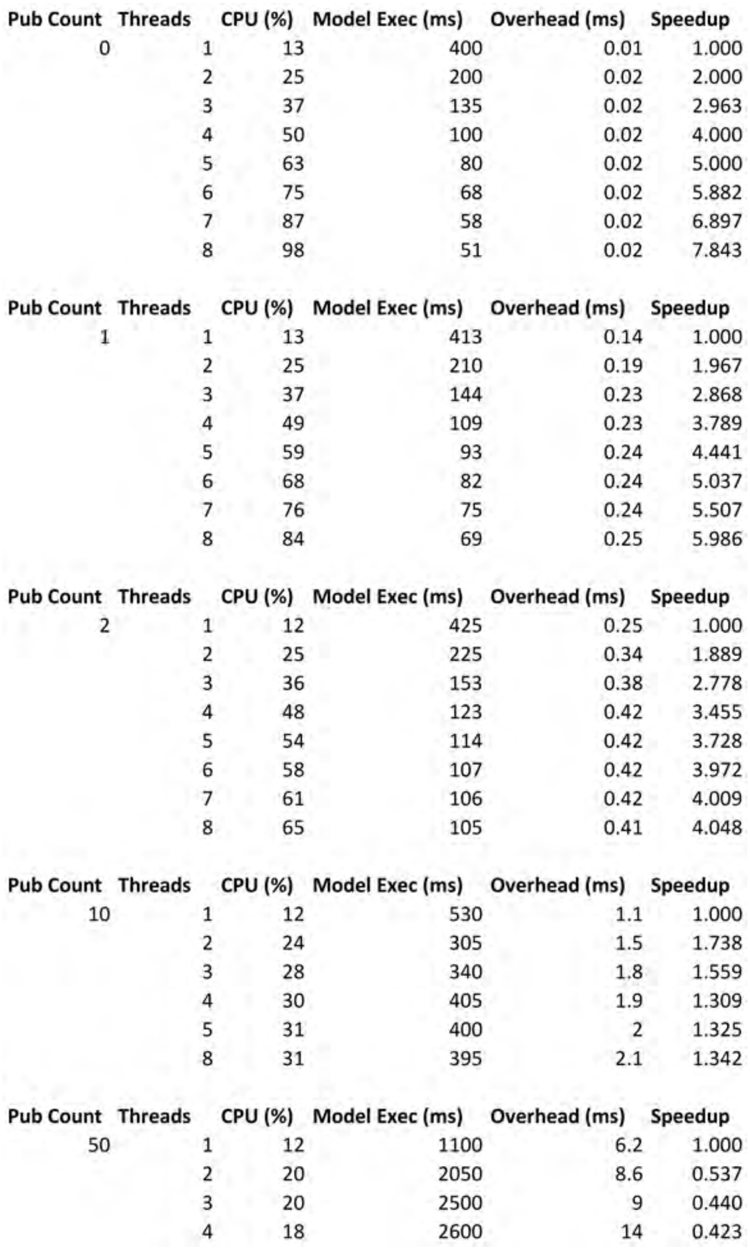
Each table in Figure 9.2 shows the following:

- the number of titles published per model per frame,
- the number of worker threads,
- the CPU utilisation,
- the time per frame it takes to run all the models,
- the time per frame it takes to read and publish titles (overhead) and
- the speedup compared to using only one worker thread.

The *model exec* time here includes the time it takes to process the received titles.

Figure 9.3 shows how well the model execution can be sped up when the models are not publishing any titles. The figure shows the thread count (worker count) on the horizontal axis and the speed increase and CPU utilisation on the vertical axis. The speed increase and CPU utilisation are very close to linear in the number of threads (i.e the work is distributed very well with almost no overhead). The CPU utilisation shows an increase from 13 to 98 percent which is *almost perfect*. Unfortunately these results are very optimistic since the models are not publishing information.

The effect of the memory manager contention can already be seen in Figure 9.4 and Figure 9.5. As more worker threads are added, the speed increase and CPU utilisation no longer increase linearly in the number of threads: backbone objects that are publishing and processing titles need access to the memory manager and contention caused by the objects being executed by multiple worker threads increases as more threads are used.



Pub Count	Threads	CPU (%)	Model Exec (ms)	Overhead (ms)	Speedup
0	1	13	400	0.01	1.000
	2	25	200	0.02	2.000
	3	37	135	0.02	2.963
	4	50	100	0.02	4.000
	5	63	80	0.02	5.000
	6	75	68	0.02	5.882
	7	87	58	0.02	6.897
	8	98	51	0.02	7.843

Pub Count	Threads	CPU (%)	Model Exec (ms)	Overhead (ms)	Speedup
1	1	13	413	0.14	1.000
	2	25	210	0.19	1.967
	3	37	144	0.23	2.868
	4	49	109	0.23	3.789
	5	59	93	0.24	4.441
	6	68	82	0.24	5.037
	7	76	75	0.24	5.507
	8	84	69	0.25	5.986

Pub Count	Threads	CPU (%)	Model Exec (ms)	Overhead (ms)	Speedup
2	1	12	425	0.25	1.000
	2	25	225	0.34	1.889
	3	36	153	0.38	2.778
	4	48	123	0.42	3.455
	5	54	114	0.42	3.728
	6	58	107	0.42	3.972
	7	61	106	0.42	4.009
	8	65	105	0.41	4.048

Pub Count	Threads	CPU (%)	Model Exec (ms)	Overhead (ms)	Speedup
10	1	12	530	1.1	1.000
	2	24	305	1.5	1.738
	3	28	340	1.8	1.559
	4	30	405	1.9	1.309
	5	31	400	2	1.325
	8	31	395	2.1	1.342

Pub Count	Threads	CPU (%)	Model Exec (ms)	Overhead (ms)	Speedup
50	1	12	1100	6.2	1.000
	2	20	2050	8.6	0.537
	3	20	2500	9	0.440
	4	18	2600	14	0.423

Figure 9.2: Parallel execution results

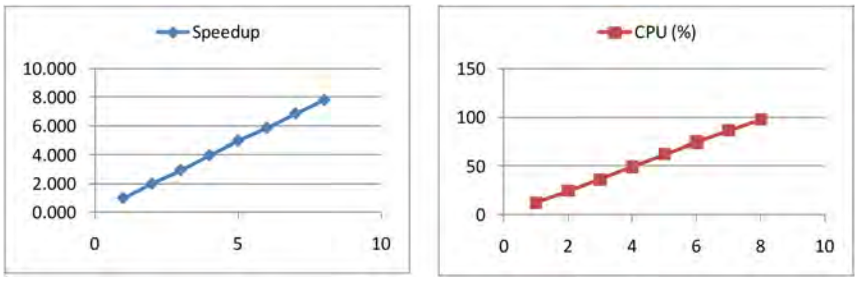


Figure 9.3: Performance with no publications

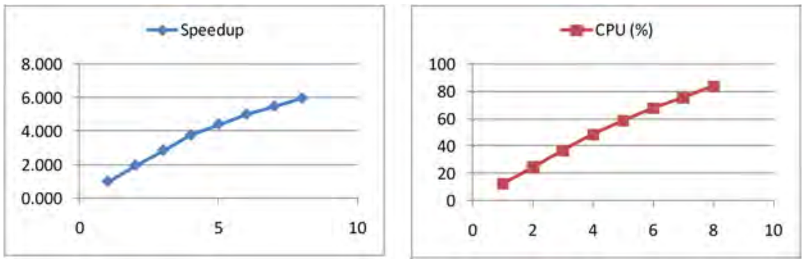


Figure 9.4: Performance with 1 title published per model per frame

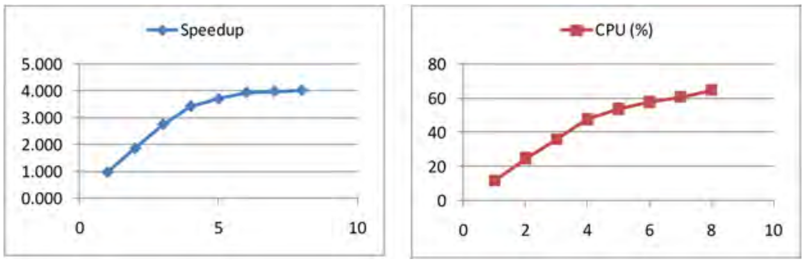


Figure 9.5: Performance with 2 titles published per model per frame

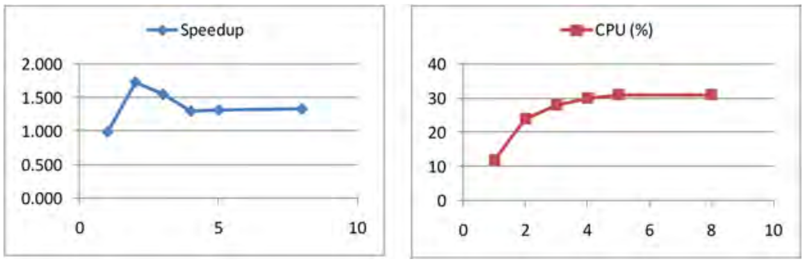


Figure 9.6: Performance with 10 titles published per model per frame

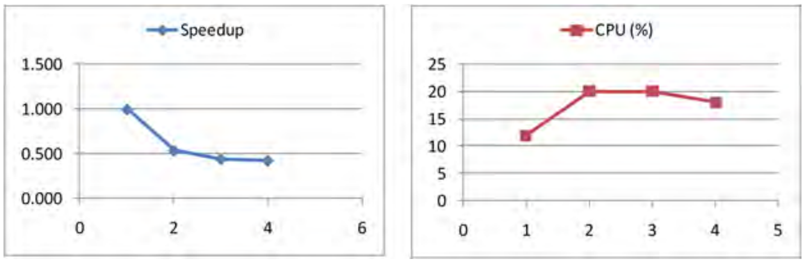


Figure 9.7: Performance with 50 titles published per model per frame

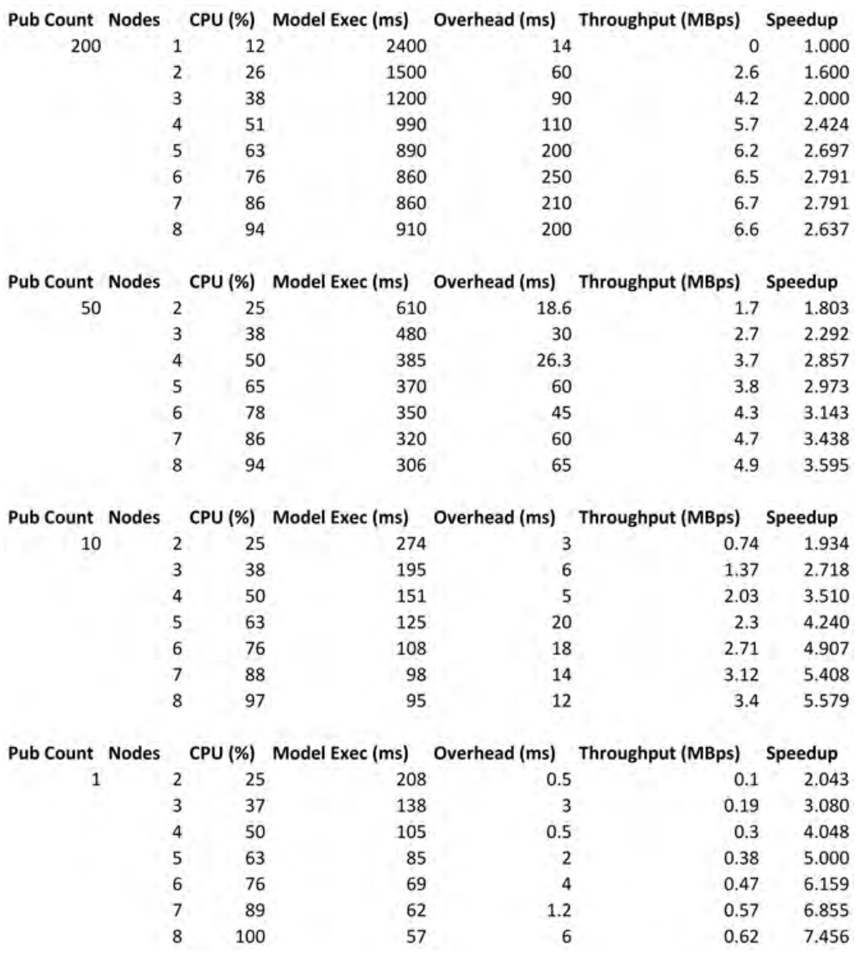
When the models publish even more titles per frame the performance suffers further. Figure 9.6 and Figure 9.7 show a decrease in performance as more worker threads are added. This indicates that the backbone objects actually spend more time waiting for access to the memory manager than they spend executing. The key to good parallel performance is minimising the number of titles published to help alleviate the contention issue between the threads.

9.1.3 Distributed Performance

This subsection discusses two sets of tests relating to distributed object execution. The first set of tests involved running multiple nodes on one of the hosts. The results are given in Figure 9.8. Each table in the figure shows the following:

- the number of titles published per model per frame,
- the number of worker threads,
- the CPU utilisation,
- the time per frame it takes to run all the models,
- the time per frame it takes to read and publish titles (overhead) and
- the total amount of data sent and received by each node (throughput in MBps), and
- the speedup compared to running all the models on only one node (see previous section).

As before, the *model exec* time here includes the time it takes to process the received titles. Each node is configured to use only one worker thread.



Pub Count	Nodes	CPU (%)	Model Exec (ms)	Overhead (ms)	Throughput (MBps)	Speedup
200	1	12	2400	14	0	1.000
	2	26	1500	60	2.6	1.600
	3	38	1200	90	4.2	2.000
	4	51	990	110	5.7	2.424
	5	63	890	200	6.2	2.697
	6	76	860	250	6.5	2.791
	7	86	860	210	6.7	2.791
	8	94	910	200	6.6	2.637

Pub Count	Nodes	CPU (%)	Model Exec (ms)	Overhead (ms)	Throughput (MBps)	Speedup
50	2	25	610	18.6	1.7	1.803
	3	38	480	30	2.7	2.292
	4	50	385	26.3	3.7	2.857
	5	65	370	60	3.8	2.973
	6	78	350	45	4.3	3.143
	7	86	320	60	4.7	3.438
	8	94	306	65	4.9	3.595

Pub Count	Nodes	CPU (%)	Model Exec (ms)	Overhead (ms)	Throughput (MBps)	Speedup
10	2	25	274	3	0.74	1.934
	3	38	195	6	1.37	2.718
	4	50	151	5	2.03	3.510
	5	63	125	20	2.3	4.240
	6	76	108	18	2.71	4.907
	7	88	98	14	3.12	5.408
	8	97	95	12	3.4	5.579

Pub Count	Nodes	CPU (%)	Model Exec (ms)	Overhead (ms)	Throughput (MBps)	Speedup
1	2	25	208	0.5	0.1	2.043
	3	37	138	3	0.19	3.080
	4	50	105	0.5	0.3	4.048
	5	63	85	2	0.38	5.000
	6	76	69	4	0.47	6.159
	7	89	62	1.2	0.57	6.855
	8	100	57	6	0.62	7.456

Figure 9.8: Distributed execution results with all nodes on one host

Figure 9.9 shows the speed increase and CPU utilisation when each model is publishing one title per frame. The figure shows the node count on the horizontal axis and the speed increase

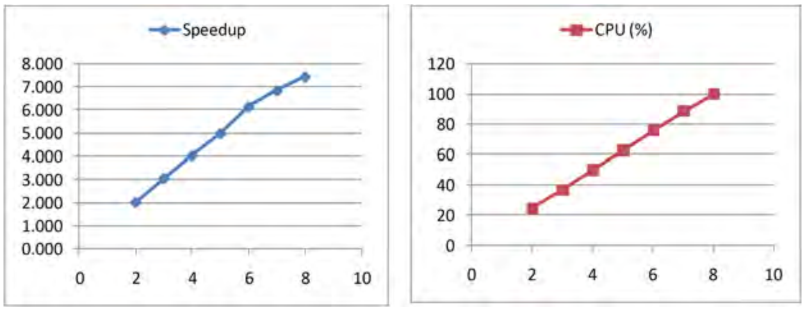


Figure 9.9: Performance with 1 title published per model per frame

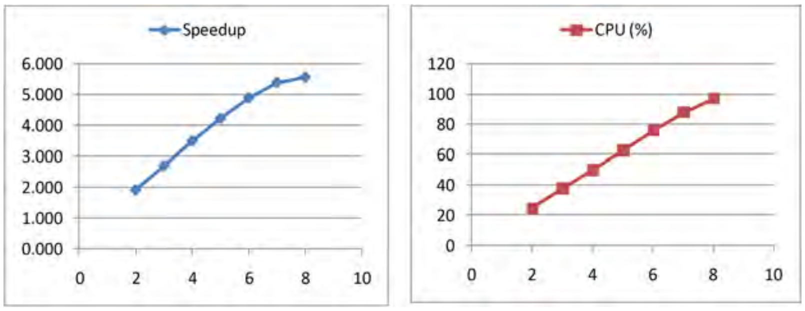


Figure 9.10: Performance with 10 titles published per model per frame

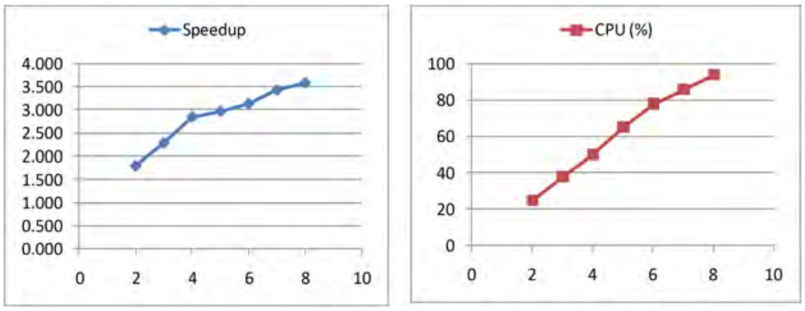


Figure 9.11: Performance with 50 titles published per model per frame

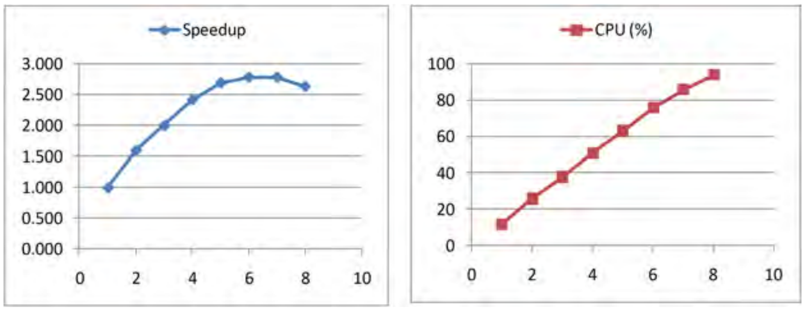


Figure 9.12: Performance with 200 titles published per model per frame

and CPU utilisation on the vertical axis. There is a linear speedup and full CPU utilisation, which is expected since the overheads are very low.

The execution performance decreases as the overhead increases (Figure 9.10), but the CPU utilisation is however still very good. Figure 9.11 shows that the speedup is less than half of what it was for the first graph, but the CPU utilisation stays very high. This is expected, since the overheads are increasing and there is no contention between the nodes (i.e the nodes are spending more time processing titles, but without having to wait for each other). When the overheads become too big the nodes start spending more time on exchanging data than they do on executing models. This results in a decrease in performance as more nodes are used (Figure 9.12). Nevertheless, the CPU utilisation still stays very high.

The second set of tests for the distributed object execution involves several hosts, running one node each. The results are given in Figure 9.13. Each table in the figure shows exactly the same data as in the previous table, namely:

- the number of titles published per model per frame,
- the number of worker threads,
- the CPU utilisation,
- the time per frame it takes to run all the models,
- the time per frame it takes to read and publish titles (overhead) and
- the total amount of data sent and received by each node (throughput in MBps), and
- the speedup compared to running all the models on only one node (see previous section).

Again, the *model exec* time here includes the time it takes to process the received titles. There is no restriction on the number of hosts that can be used (as opposed to the previous tests where only eight processor cores were available per host). Each node is configured to use only one worker thread and with only one node per host, the CPU utilisation stays constant.

Figure 9.14 shows the speed increase and network throughput when each model is publishing 50 titles per frame. The figure shows the node count on the horizontal axis and the speed increase and network throughput on the vertical axis. Figures 9.14 and 9.15 clearly show that distributing the model execution among several hosts increases the performance. It actually performs better than expected, since it outperformed the *local host* case (see Figure 9.8) which was expected to be faster. It appears that the *local host* TCP/IP capability of the host operating system is slower than expected.

The *model execution* time increases when more data is published (Figure 9.13). This is expected since the model execution time includes the processing of the received titles. The node throughput however changes very little when more data is published. This indicates that the node throughput is at its maximum and that the execution performance is limited by the sending, receiving and processing of titles rather than the actual model execution. Figure 9.17 shows the maximum throughput achieved when the test model is modified to have an execution time of 0ms. The maximum values are very close to what is shown in Figure 9.13, further indicating that the execution performance is limited by the processing of titles rather than the actual model execution. The key here to good distributed performance is minimising the data exchange between hosts.

Pub Count	Nodes	CPU (%)	Model Exec (ms)	Overhead (ms)	Throughput (MBps)	Speedup
400	1	12	4100	50	0	1.000
	2	12	3250	120	2.4	1.262
	3	12	2250	150	4.5	1.822
	4	12	1700	200	6.5	2.412
	5	12	1450	170	7.9	2.828
	6	12	1220	180	9.5	3.361
	7	12	1050	220	10.9	3.905
	8	12	940	230	12	4.362
	9	12	860	240	13.1	4.767
	10	12	780	260	14.2	5.256

Pub Count	Nodes	CPU (%)	Model Exec (ms)	Overhead (ms)	Throughput (MBps)	Speedup
200	2	12	1675	55	2.4	1.433
	3	12	1180	80	4.4	2.034
	4	12	880	110	6.2	2.727
	5	12	735	110	7.7	3.265
	6	12	630	110	9.2	3.810
	7	12	550	130	10.5	4.364
	8	12	490	140	11.5	4.898
	9	12	460	130	12.6	5.217
	10	12	420	140	13.5	5.714

Pub Count	Nodes	CPU (%)	Model Exec (ms)	Overhead (ms)	Throughput (MBps)	Speedup
50	2	12	520	15	1.9	2.115
	3	12	360	30	3.6	3.056
	4	12	280	35	5	3.929
	5	12	230	30	6.3	4.783
	6	12	190	35	7.8	5.789
	7	12	165	40	8.6	6.667
	8	12	145	100	9	7.586
	9	12	135	40	10.5	8.148
	10	12	120	40	11.6	9.167

Figure 9.13: Distributed execution results with one node per host

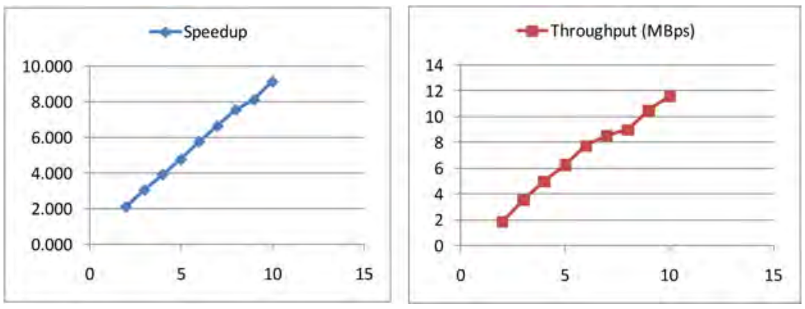


Figure 9.14: Performance with 50 titles published per model per frame

9.2 Application Examples

The previous section discussed the general M&S performance of the framework. This section discusses applications which served to further verify some of the interoperability and virtualisation (i.e. M&S) capabilities of the implemented framework. Four applications within the C2 domain were developed and successfully applied. In addition, three test applications were also developed. These test applications served to verify the behaviour and performance of the *backbone* and *infrastructure* layer implementations of the framework. The C2 applications are more complex and their successful implementation provides additional

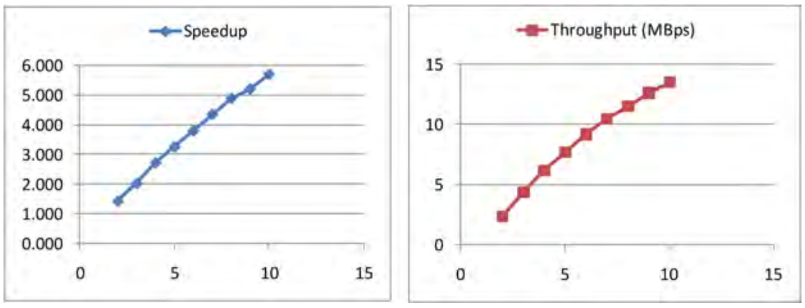


Figure 9.15: Performance with 200 titles published per model per frame

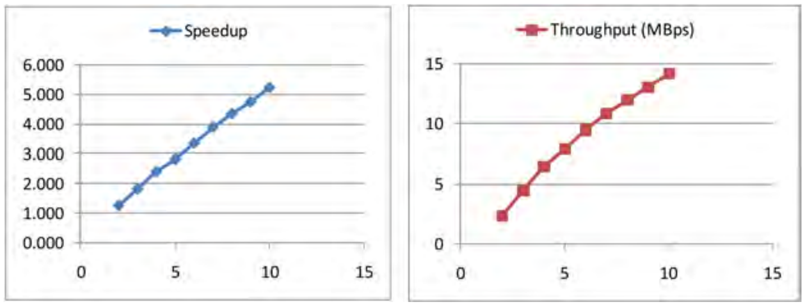


Figure 9.16: Performance with 400 titles published per model per frame

Pub Count	Nodes	Model Exec (ms)	Overhead (ms)	Throughput (MBps)
200	2	1480	51	2.67
	3	1040	70	4.8
	4	780	100	7

Figure 9.17: Performance with a model exec time of 0ms

positive evidence in verifying the *interoperability* and *application* layers of the framework.

The framework and the three test applications were designed and implemented by the author. However, the four C2 applications discussed below were developed by the relevant CSIR project teams in which the author was merely the technical lead[1]. Although the *development* of the C2 applications cannot be regarded as part of the research discussed in this dissertation, the fact that they were successfully implemented using the framework bears testimony to its validity and usability and, for this reason, deserves to be mentioned here.

The test applications are:

- the performance test application discussed in the previous section of this chapter,
- a test application that simulates the flocking behaviour of birds, and
- an implementation of Conway's game of life.

The C2 applications are:

[1]Because these C2 applications deal with restricted information they are only discussed in broad overview. The author can be contacted for additional information.

- an air to air tactics evaluation tool for fighter aircraft,
- a protocol gateway that facilitated air force, navy and military system interoperability during preparatory field exercises for the soccer world cup 2010,
- a radar emulator for adding additional information sources to an air force system, and
- a *joint operations operator console* concept demonstrator.

The C2 applications all use custom user interface and 2D/3D visualisation layers that are integrated with the framework. These layers use technologies like *wxWidgets*, *Qt* and *Open Scene Graph*. The user interface and visualisation layers are application specific and do not form part of the research effort discussed in this dissertation. The previous chapter (Chapter 8) did however discuss how such application specific layers integrate with the framework.

The framework source code has been included on the DVD accompanying this dissertation. The complete source code for the three test applications is also included [2].

9.2.1 A Simulation of Flocking Behaviour

Craig Reinolds developed an artificial life program called *Boids* in 1986 which simulates the flocking behaviour of birds. The test application discussed in this section implements that same *boid* behaviour. The flocking behaviour is simulated by creating multiple instances of a model (or *boid*) that moves around according to a fixed set of rules. The rules define how each boid behaves within its flock: each boid tries to stay close to the center of the flock; each boid tries to move in the same direction as the flock; and, each boid tries to avoid flying into other boids. It is possible to create very realistic flocking behaviour with these simple rules.

In order to test the *infrastructure* layer the flocking behaviour is simulated in a spherical coordinate system (i.e. the boid positions are represented using latitude, longitude and altitude). The model rules however operate within a local cartesian coordinate system and the infrastructure layer contains the components that represent the two coordinate systems and can translate between them.

Each boid subscribes to the state of every other boid and also publishes its own state using a state title. The state title provides the position (in spherical coordinates) and the velocity of a boid. Each boid builds up a set of flock-mate positions (in local coordinates) with the state titles from its subscriptions. Each boid then calculates its own velocity based on the *boid* rules, updates its own state and then publishes it. This type of simulation, with the models operating at discrete time steps, can be referred to as discrete time simulation (DTS).

The boid models follow smooth trajectories that can be approximated by some form of prediction. The framework accommodates techniques like *dead-reckoning* by allowing subscribers to predict the future state of the information received from publishers. The boid model updates the set of flock-mate positions with state titles that come through on the subscriptions or by predicting the last known state for flock-mates that did not update. The boid model does not publish its own state if it knows that the prediction used by other boids

[2]The four C2 applications created with the framework can not be included on the DVD since the applications contain restricted or sensitive information.

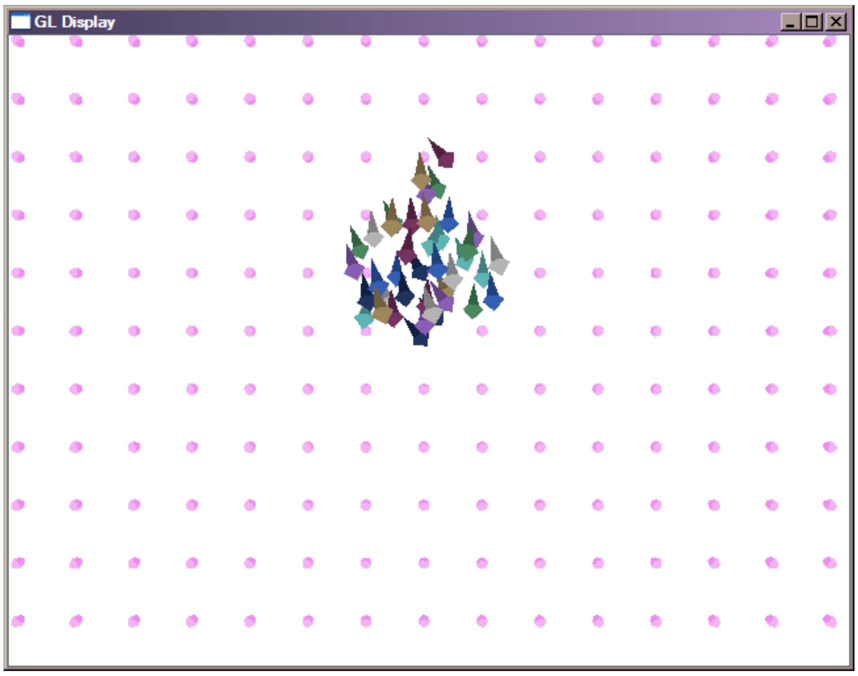


Figure 9.18: A 3D View of 40 Flocking Models in Test Application

would produce an accurate enough result (this functionality is part of the framework). This can be used to make a trade off between accurate boid movement and publishing fewer titles.

Figure 9.18 shows a simple custom 3D view that was created as part of this test application. This instance had 40 models specified in the XML scenario file. The boid model execution can also be distributed and parallelised in exactly the same way as in the *performance test application* discussed in the previous section of this chapter.

This test application simulated the flocking behaviour as expected and ran without problems. The application can be found on the accompanying DVD.

9.2.2 Conway's Game of Life

John Horton Conway developed *Conway's Game of Life* in 1970. The game has no players and takes place on a two dimensional grid of square cells. The cells switch on and off based on specific game rules and the game grid evolves based on the initial cell pattern. The game runs through multiple iterations, each cell changing in each evolution step on the basis of rules relating to the cell states of its immediate neighbours.

Each cell has eight immediate neighbours (cells that are directly vertically, horizontally and diagonally adjacent). The rules are:

- a cell that is off and has exactly three on neighbours switches on in the next iteration of the game;
- a cell that is on and has more than three on neighbours switches off in the next iteration;
- a cell that is on and has less than two on neighbours switches off in the next iteration.

The rules aim to mimic the behaviour of cells living and dying as a consequence of reproduction, overcrowding and under-population. The test application discussed in this section implements the game of life with one model per cell subscribing to the state of its eight neighbours and publishing its own state whenever it changes.

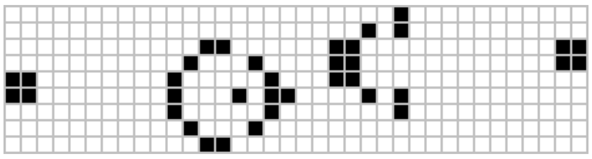


Figure 9.19: The Initial Cell Pattern Used by the Test Application

Figure 9.19 shows the initial pattern used by the test application: the cell models are created in either the on state or in the off state based on this pattern. This particular pattern, when being evolved by the game of life, is called the *Gosper Glider Gun* and the cells switch on and off in such a way that it looks like a gun firing bursts that fly off indefinitely.

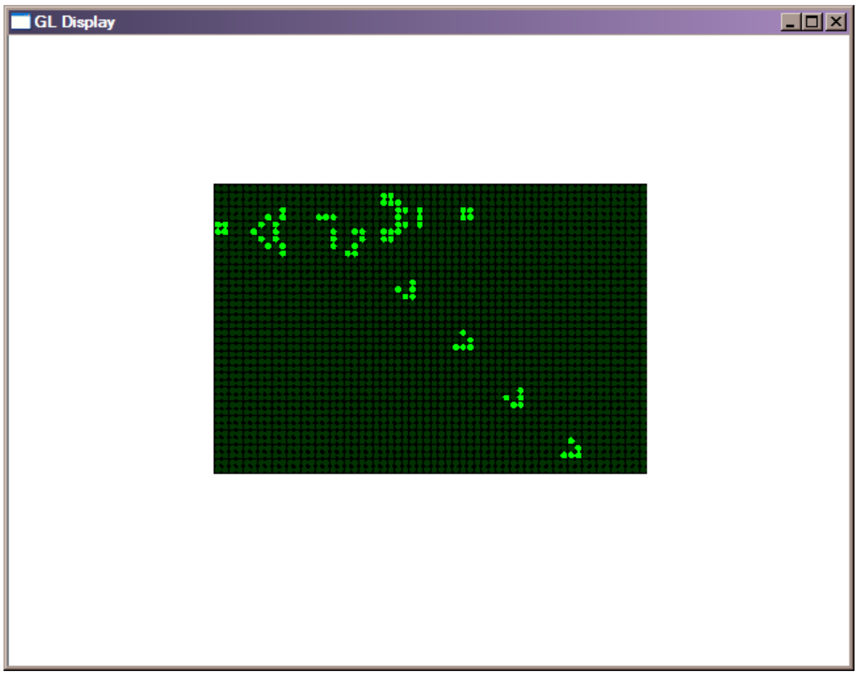


Figure 9.20: A View of 2000 Cells in Test Application

Each cell in the test application publishes its initial state and then only publishes its state again when it changes. Each cell also remembers the last state of each of its neighbours in case they do not publish. The test application uses a 50 by 40 cell grid—that is 2000 cell models running in the backbone (shown in Figure 9.20). The performance is still extremely good since each cell only reacts if it gets a new update from one of its neighbours and then only publishes its own state if its state changes. This type of simulation, where the models only react to events and only publish events, is referred to as discrete event simulation.

The cell model execution can also be distributed and parallelised in exactly the same way as in the *performance test application* discussed in the previous section of this chapter. This application can be found on the accompanying DVD. It demonstrates how scalable the backbone object execution is—it easily runs 2000 models.

9.2.3 An Tactics Evaluation Tool for Fighter Aircraft

In 2008 an early version of the framework implementation was used to develop an air-to-air tactics evaluation tool for the new generation Gripen fighter aircraft acquired by the South African Airforce (SAAF). The Gripen has a higher situational awareness than previous SAAF aircraft and it also has the ability to share information with other aircraft over a tactical data link.

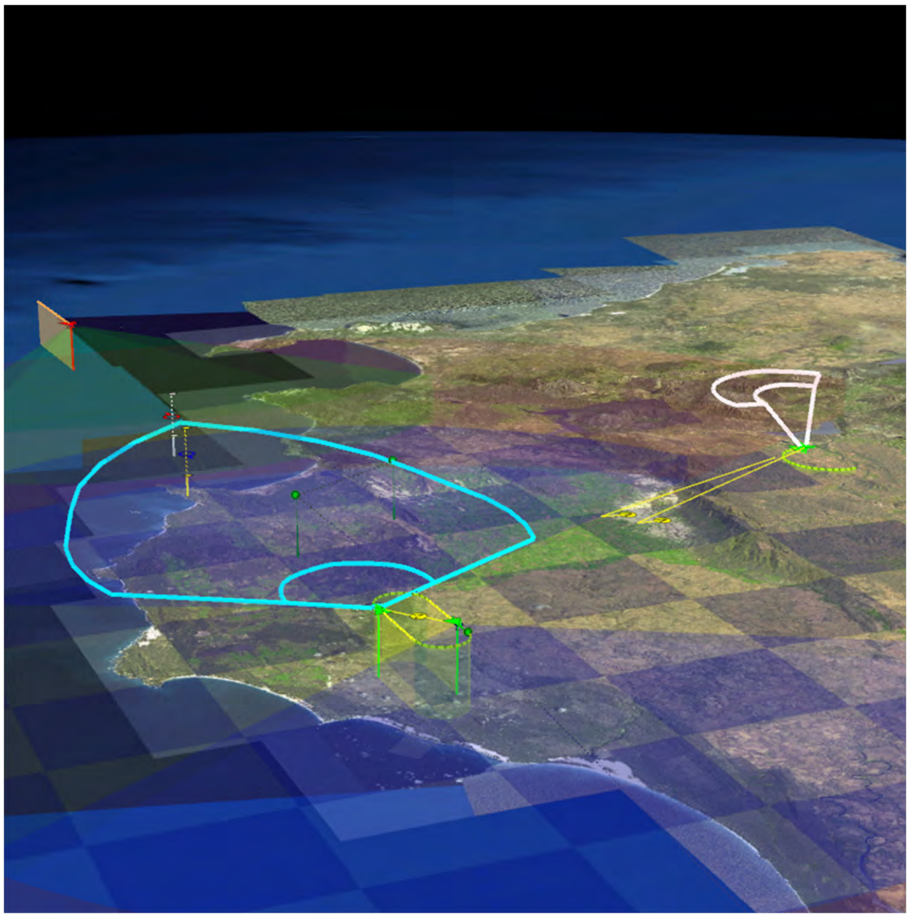


Figure 9.21: The Tactics Evaluation Tool (3D view)

The goal of the tool was to help develop new air-to-air tactics for the Gripen aircraft using modelling and simulation (M&S). The key is simulating accurate aircraft flight paths, correctly simulating the behaviour of the tactical data link between aircraft and correctly simulating the behaviour of the aircraft radar. The tool helps the SAAF to quickly generate information regarding the aircraft performance. This information can then be applied to help the SAAF use the official Gripen mission planning systems more effectively.

The tool is currently still being extended by the Defence Peace Safety and Security, Aeronautics research group within the CSIR. A custom 3D view component, developed by the CSIR, is also integrated into the tool in order to visualise various aspects of the aircraft,

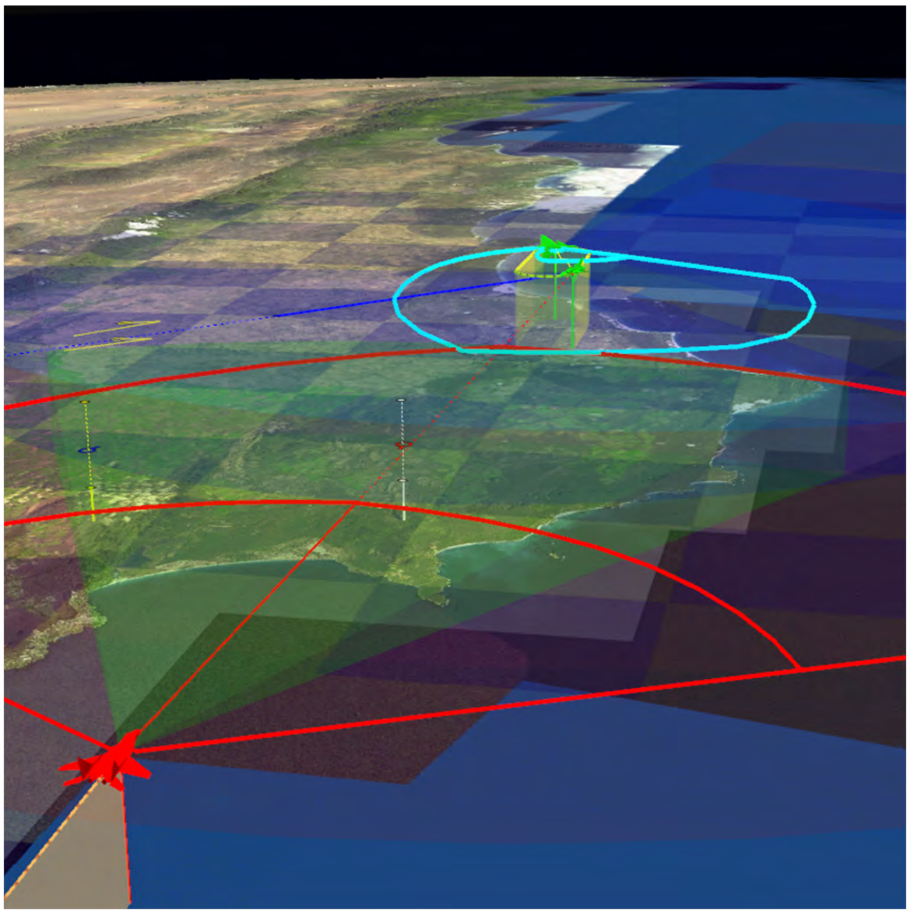


Figure 9.22: The Tactics Evaluation Tool (Closeup of Aircraft Model)

radar and data link models. Figure 9.21 and Figure 9.22 show what the tool looks like.

9.2.4 A Command and Control Protocol Gateway

The framework was used to build a gateway application that can act as a message router for various systems and simulators (Duvenhage and Terblanche, 2008). The gateway implements all the links required to connect to the relevant systems and to exchange information with these systems. The gateway also translates the information to and from an internal representation. This allows the gateway to route information between systems, acting as a C2 hub. Most of the gateway functionality is implemented in the *infrastructure* and *interoperability* layers of the framework, giving all applications access to it. The gateway consists of multiple link objects and one gateway object.

The gateway links are implemented as different backbone objects that publish the information they receive. The link objects are reusable across multiple applications. The links are fault tolerant and translate the information of the various external systems to a unified internal representation that is used throughout the rest of the application.

The gateway object subscribes to all the links; it routes and filters information; and it publishes back to the links. The links also subscribe to information from the gateway. The subscriptions and publications are registered in such a way that the gateway object can receive information from any link object and publish back to specific link objects based on the routing rules. The gateway implements special *internal* links that enable filtering and exchanging of information with other backbone objects. The internal links allow the gateway to be part of any application created with the framework. The routing rules, links and internal links are setup in the XML scenario file.

This gateway was used extensively to facilitate system interoperability during preparatory military field exercises for the Soccer World Cup 2010—the police, air force, army and navy had to work together and system interoperability was crucial. The gateway was used to relay tactical information between operational air force, navy and army systems. This helped create awareness of the importance of interoperability for joint operations.

9.2.5 A Radar Emulator

The gateway also has the ability to emulate (i.e. virtualise) a specific type of radar system by implementing the same link protocol (in a link object) as an actual radar system does. Several radar systems used by the South African Air Force (SAAF) use this protocol.

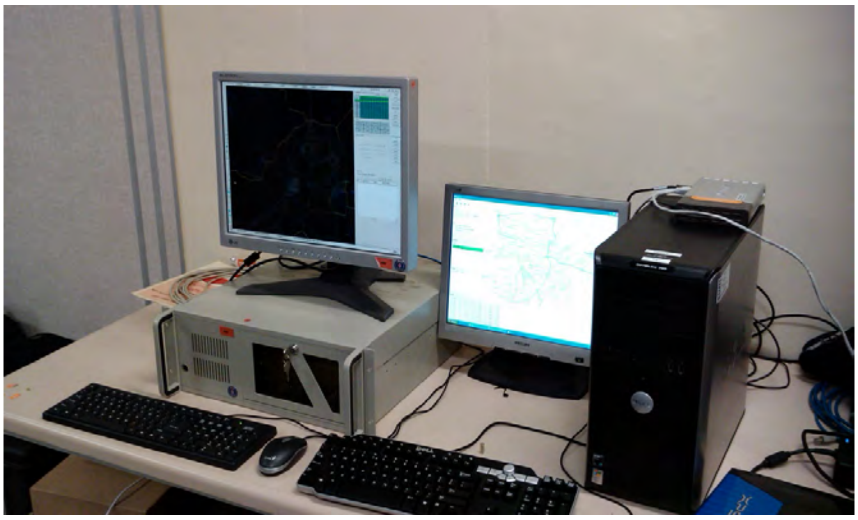


Figure 9.23: The Radar Emulator Test Setup at the SAAF Head Quarters

Figure 9.23 shows a test setup of this *radar emulator* at the SAAF Head Quarters, Pretoria, South Africa. The machine running the gateway/emulator software is on the right and a stand-alone SAAF air picture display system is on the left. The SAAF system accepts radar inputs via an HDLC interface card. The *emulator* machine also has an HDLC interface card and connects to the SAAF system via a serial cable (as the real system would). For all intents and purposes the *emulator* machine then looks like a real radar system to the SAAF system.

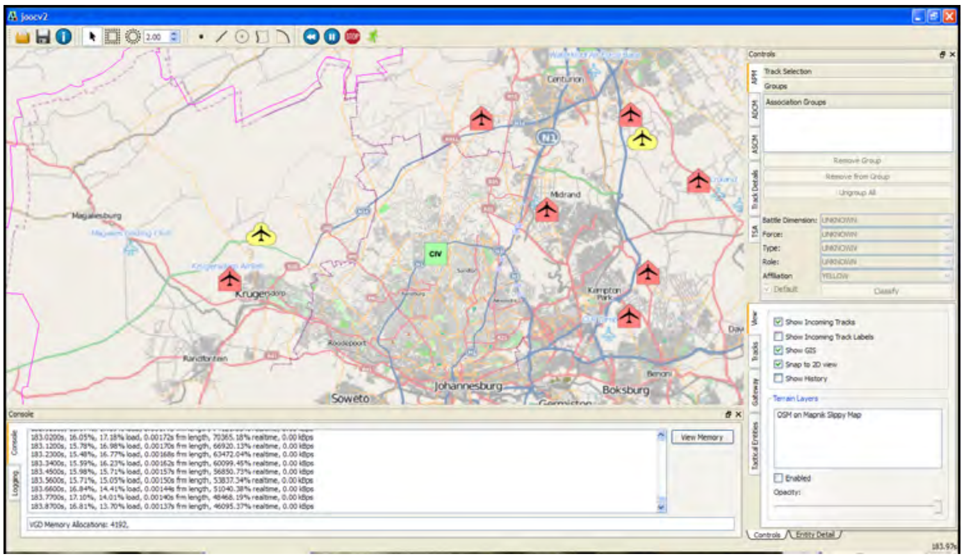
The radar emulator has been used during SANDF field trials to integrate additional sensors into a specific SAAF system. The gateway translates information from systems that would normally not function with the SAAF system into something that looks like information from one of the standard SAAF radars. This gives operators access to additional information that would otherwise not have been so readily available.

In the test setup shown in Figure 9.23 the gateway listens in on a SAAF aircraft tactical radio link (using a compatible link radio) and receives reports of each aircraft's own position as well as contact reports from each aircraft's onboard radar. These reports are then sent to the SAAF air picture display system as radar plots. This test setup was built in collaboration with Saab Systems South Africa who provided all the relevant protocol specifications, the HDLC card interface and the radio interface. Saab also assisted with setting up the radio link to the aircraft.

9.2.6 A Joint Operations Operator Console

The Joint Operations Operator Console (JOOC) is a generic platform for technology demonstrators within the Joint Command and Control (JC2) context. The JOOC demonstrates concepts concerning *air picture management*, *multi-sensor fusion* and system interoperability. *Air picture management* concerns sensor tracks of aircraft and the management thereof—an air picture manager can classify or modify existing tracks to be more accurate. Multi-sensor fusion is the process whereby tracks from two or more sensors are associated and combined into a single set of tracks (i.e. only one track for each aircraft in the air).

The JOOC includes the C2 Protocol Gateway capabilities as well as a 3D view for geospatial information and additional services that enable air picture management and multi-sensor fusion. A geospatial view displays the situational picture and allows the user to interact with it.



Battle Dimension: UNKNOWN
Force: UNKNOWN
Type: UNKNOWN
Role: UNKNOWN
Affiliation: YELLOW
Default

[✓] Show Incoming Tracks
[ ] Show Incoming Track Labels
[✓] Show GIS
[✓] Snap to 2D view
[ ] Show History

Terrain Layers
OSM on Mapnik Slippy Map
[ ] Enabled
Opacity:

```
183.0200s, 16.05%, 17.18% load, 0.00172s frm length, 70365.18% realtime, 0.00 kBps
183.1200s, 15.78%, 16.98% load, 0.00170s frm length, 66920.13% realtime, 0.00 kBps
183.2300s, 15.48%, 16.77% load, 0.00168s frm length, 63472.04% realtime, 0.00 kBps
183.3400s, 15.59%, 16.23% load, 0.00162s frm length, 60099.45% realtime, 0.00 kBps
183.4500s, 15.98%, 15.71% load, 0.00157s frm length, 56850.73% realtime, 0.00 kBps
183.5600s, 15.71%, 15.05% load, 0.00150s frm length, 53837.34% realtime, 0.00 kBps
183.6600s, 16.84%, 14.41% load, 0.00144s frm length, 51040.38% realtime, 0.00 kBps
183.7700s, 17.10%, 14.01% load, 0.00140s frm length, 48468.19% realtime, 0.00 kBps
183.8700s, 16.81%, 13.70% load, 0.00137s frm length, 46095.37% realtime, 0.00 kBps
```

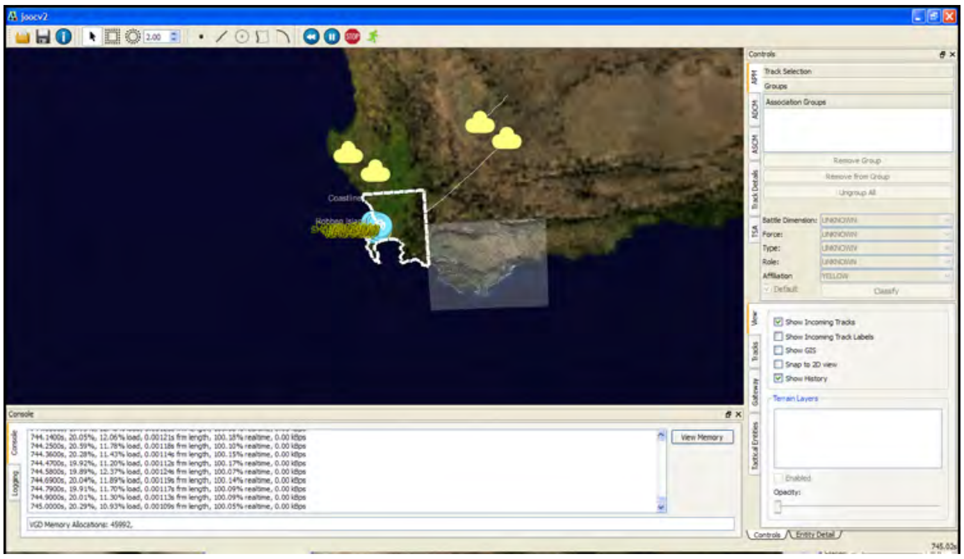
VGD Memory Allocations: 4192.

183.97s

Figure 9.24: The JOOC with Various Air Tracks in the View

The JOOC is currently being extended for the Ground Based Air Defence (GBAD) environment. It will be used to test concepts in regimental-level air defence (a role of the South African Army). Regimental-level air defence concerns the control and management of multiple air defence deployments, optimising air defence by looking at all the deployed equipment and resources in a holistic fashion and collating the air defence efforts.

The extended JOOC (shown in Figure 9.24 and figure 9.25) will allow one to set up different mock-up air defence terminals and have real military personnel experiment with and evaluate



```
744.1400s, 20.05%, 12.06% load, 0.00121s frm length, 100.18% realtime, 0.00 kBps
744.2500s, 20.59%, 11.78% load, 0.00118s frm length, 100.10% realtime, 0.00 kBps
744.3600s, 20.28%, 11.43% load, 0.00114s frm length, 100.15% realtime, 0.00 kBps
744.4700s, 19.92%, 11.20% load, 0.00112s frm length, 100.17% realtime, 0.00 kBps
744.5800s, 19.89%, 12.37% load, 0.00124s frm length, 100.07% realtime, 0.00 kBps
744.6900s, 20.04%, 11.89% load, 0.00119s frm length, 100.14% realtime, 0.00 kBps
744.7900s, 19.91%, 11.70% load, 0.00117s frm length, 100.09% realtime, 0.00 kBps
744.9000s, 20.01%, 11.30% load, 0.00113s frm length, 100.09% realtime, 0.00 kBps
745.0000s, 20.29%, 10.93% load, 0.00109s frm length, 100.05% realtime, 0.00 kBps
```

VGD Memory Allocations: 45992.

Battle Dimension: UNKNOWN
Force: UNKNOWN
Type: UNKNOWN
Role: UNKNOWN
Affiliation: YELLOW

[✓] Show Incoming Tracks
[ ] Show Incoming Track Labels
[ ] Show GIS
[ ] Snap to 2D view
[✓] Show History

Figure 9.25: The JOOC With Some Air Tracks and GBADS Elements

their operational procedures for regimental fire control. What the role of each air defence operator should be and how different operators should interact with each other, could then be optimised.

Figure 9.24 and Figure 9.25 show the JOOC user interface. In both figures, the geospatial view is in the top-center panel, and the main user interface panels are at the bottom and to the right. The bottom panel contains tabs for logging and a text based console that reports framework event and status information—the M&S capability of the framework is always active. The two panels to the right show the gateway routes and link status, some view controls and several tabs related to air defence control. The panels are user interface widgets that use *control* objects (see Chapter 8) to exchange the relevant information and events with backbone objects.

This extended JOOC will also be used to evaluate possible ways of integrating the GBAD systems with air defence systems from the South African Air Force (SAAF) and the South African Navy (SAN). Integration of the Army, Air Force and Navy air defence capabilities is referred to as Joint Air Defence (JAD).

Figure 9.26 shows an earlier version of the JOOC that was integrated with the base station of a military UAV during a demonstration at the CSIR. The integration was done via the gateway that forms part of the JOOC. The UAV position as well as the positions of targets of interest could be sent from the UAV's base station (via a serial interface) to the JOOC and displayed in 3D. The gateway also has a link that can get civilian air traffic information from local air traffic control centres. This demonstration verified that it is possible to integrate and present military UAV and civilian air traffic information on one view.

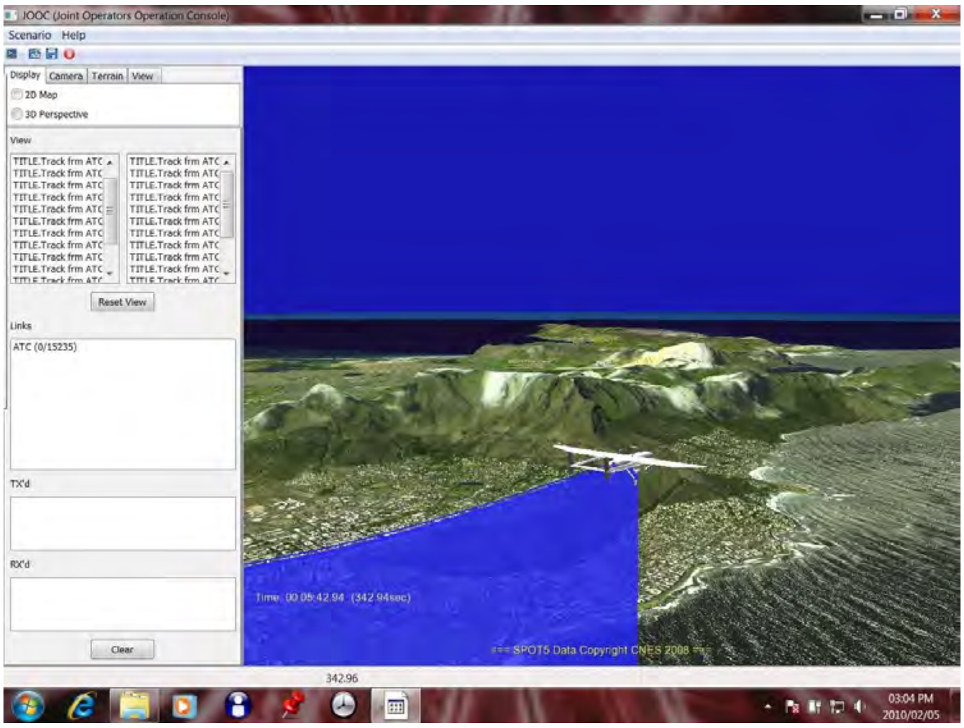


Figure 9.26: The JOOC Integrated With a Military UAV Base Station

9.3 Formal Evaluation

Part two of this dissertation showed how to describe the structure and behaviour of software. In this section, critical components of the software framework design are identified and then formally described and evaluated using the methods discussed in Chapter 4.

9.3.1 Distributed Execution

Distributed model execution involves executing backbone objects on different nodes and transferring data between nodes. The backbone object execution is divided up into frames and the nodes operate in a lock-step fashion (i.e. a node will not start a new frame until all the other nodes have finished the previous frame). Each frame goes through several steps or states which can be summarised as follows:

1. The node receives communications events, generated in the previous frame, from all other nodes (the information from each node is abstracted into one event). The node will not continue until it has received communications events from all nodes.
2. The node processes the communications events (delivering the issues generated in the previous frame to all backbone objects; and then executing the relevant backbone objects).
3. The node sends out the new information published by the objects and generated by the backbone as communications events to all the other nodes.
4. The node goes back to step one and starts waiting for communications events from

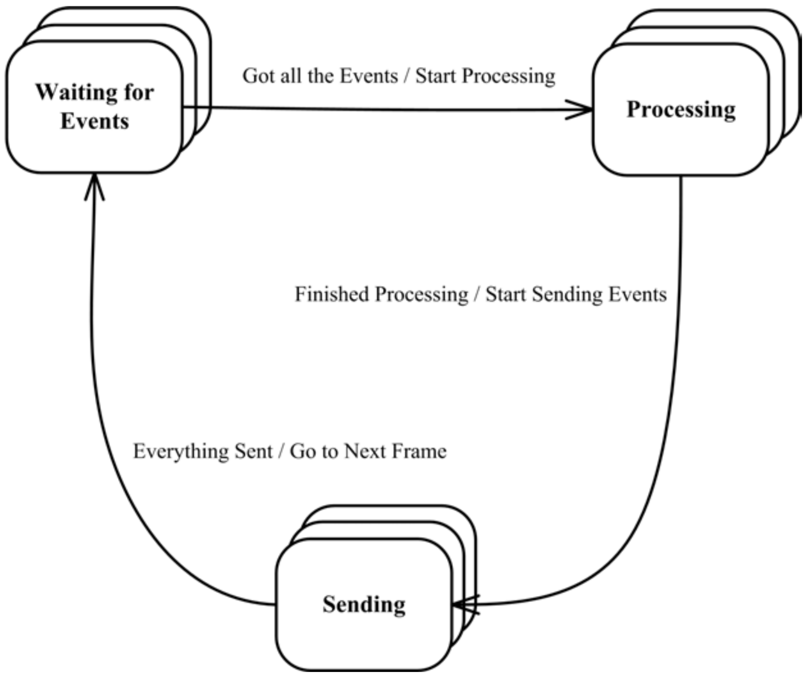

Waiting for Events
Got all the Events / Start Processing
Processing
Finished Processing / Start Sending Events
Everything Sent / Go to Next Frame
Sending

Figure 9.27: A Finite State Machine Showing the States of a Node Frame

other nodes. This ignores the additional wait the framework can make to keep the simulation from running faster than real-time.

These steps match the backbone object execution steps discussed in Chapter 8, but have been adapted for this discussion. Figure 9.27 shows the FSM for a node frame (the layered blocks indicate that multiple nodes can be in that state). The start of the FSM is also not shown, since the very first frame of each node does not include the *read* step—this FSM excludes the first frame.

The following CSP analysis provides an alternative view to the FSM shown in Figure 9.30. Figure 9.28 shows the communication events between three nodes. The nodes are numbered 1 to 3 (*NODE1*, *NODE2*, *NODE3*) and each node can be seen as a process reading and writing communications events. The communication events are *C12, C13, C21, C23, C31 and C32.* To make the analysis simpler the events are reduced to *C1, C2, C3* for the events that are received from the nodes and *D1, D2 and D2* for the events that are sent from the nodes. It is correct to rename the sent events from *C* to *D*, since sent events are only read in the next frame. The CSP for the individual processes can be written as follows:

$$\begin{aligned} NODE1 &= (C2 \rightarrow C3 \rightarrow D1 \rightarrow NODE1) \mid \\ &\quad (C3 \rightarrow C2 \rightarrow D1 \rightarrow NODE1) && (9.1)\\ NODE2 &= (C1 \rightarrow C3 \rightarrow D2 \rightarrow NODE2) \mid \\ &\quad (C3 \rightarrow C1 \rightarrow D2 \rightarrow NODE2) && (9.2)\\ NODE3 &= (C1 \rightarrow C2 \rightarrow D3 \rightarrow NODE3) \mid \\ &\quad (C2 \rightarrow C1 \rightarrow D3 \rightarrow NODE3) && (9.3)\end{aligned}$$

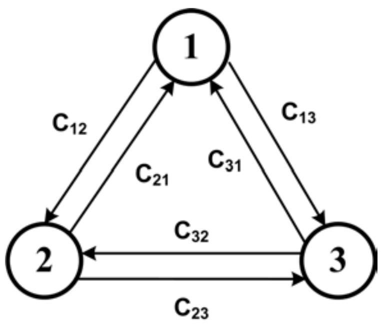


Figure 9.28: The Communication Events Between Three Nodes

The processes run in parallel, but have to synchronise on common events. This can be expressed as:

$$NODES \quad = \quad NODE1 \parallel NODE2 \parallel NODE3 \tag{9.4}$$

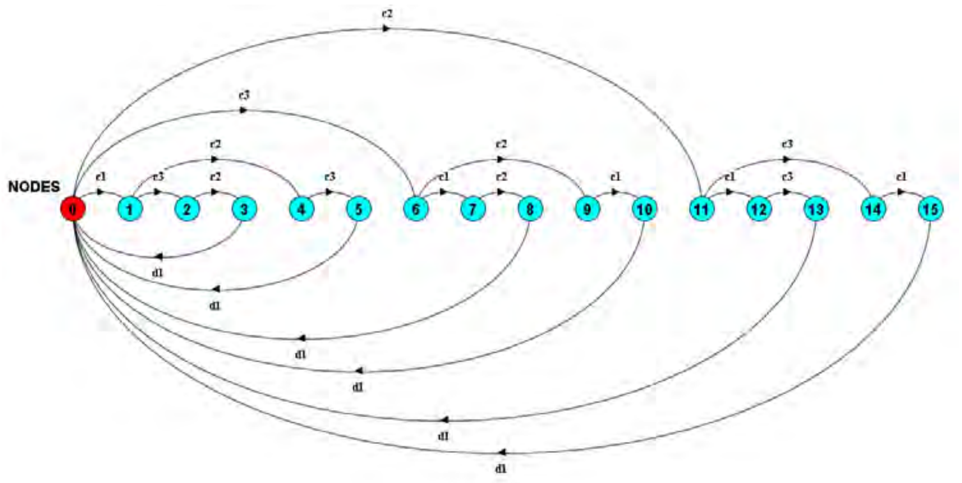


Figure 9.29: The LTSA transition diagram for *NODES*

This CSP model was evaluated using a tool called the *Labelled Transition System Analyser (LTSA)*. The tool uses a Finite State Process (FSP) textual notation to represent the CSP. The tool can analyse the models for deadlock. The CSP model for *NODES* can be expressed in FSP as:

$$\begin{aligned} NODE1 &= (c2- > c3- > d1- > NODE1 \mid \\ & \quad c3- > c2- > d1- > NODE1). \quad (9.5) \\ NODE2 &= (c1- > c3- > d2- > NODE2 \mid \\ & \quad c3- > c1- > d2- > NODE2). \quad (9.6) \\ NODE3 &= (c1- > c2- > d3- > NODE3 \mid \\ & \quad c2- > c1- > d3- > NODE3). \quad (9.7) \end{aligned}$$

$$||NODES = (NODE1 \parallel NODE2 \parallel NODE3)/\{d1/d3, d1/d2\}. \quad (9.8)$$

The events *d2* and *d3* are renamed to *d1* in the FSP model for *NODES*. This is legitimate, since the order in which the different nodes write is not important (the nodes only have to write something after the reads). The model forces synchronisation on the write and on the reads. Using the LTSA tool it was determined that there is no possibility of deadlock. The transition diagram generated for *NODES* by the LTSA tool is also shown in Figure 9.29.

9.3.2 The Frame Execution and Multi-threading

The backbone objects can be executed concurrently (i.e. by multiple worker threads), since backbone objects do not interact directly with each other. The backbone has to interact with the *node hub* to send and received information. This can only happen on a single worker thread, called the *main thread*, since the node hub implementations might not be thread safe (see Chapter 8). Because of this the other worker threads have to block or wait while the main thread does the extra work. The *Distributed Object Execution* is discussed in detail in Chapter 8.

The states of each node frame can be summarised as follows:

1. The main thread receives communications events from all the other nodes via the node hub (these communications events were generated in the previous frame). The main thread will not continue until it has received communications events from all nodes. The worker threads wait for the main thread to continue.
2. The worker threads (including the main thread) process the communications events. The threads are responsible for delivering all issues published in the previous frame and executing the relevant objects. The threads can all work on the same set of issues, since the issues are not modified while delivered.
3. Once all the threads have finished processing, the threads continue. The main thread collects all the new information published by the backbone objects and then sends the information as new communications events to all the other nodes. The worker threads are finished until the next frame and start waiting for the main thread.
4. The node goes back to step one and starts waiting for communications events from other nodes. This ignores the additional wait the framework can make to keep the simulation from running faster than real-time.

These steps also match the backbone object execution steps discussed in Chapter 8, but have

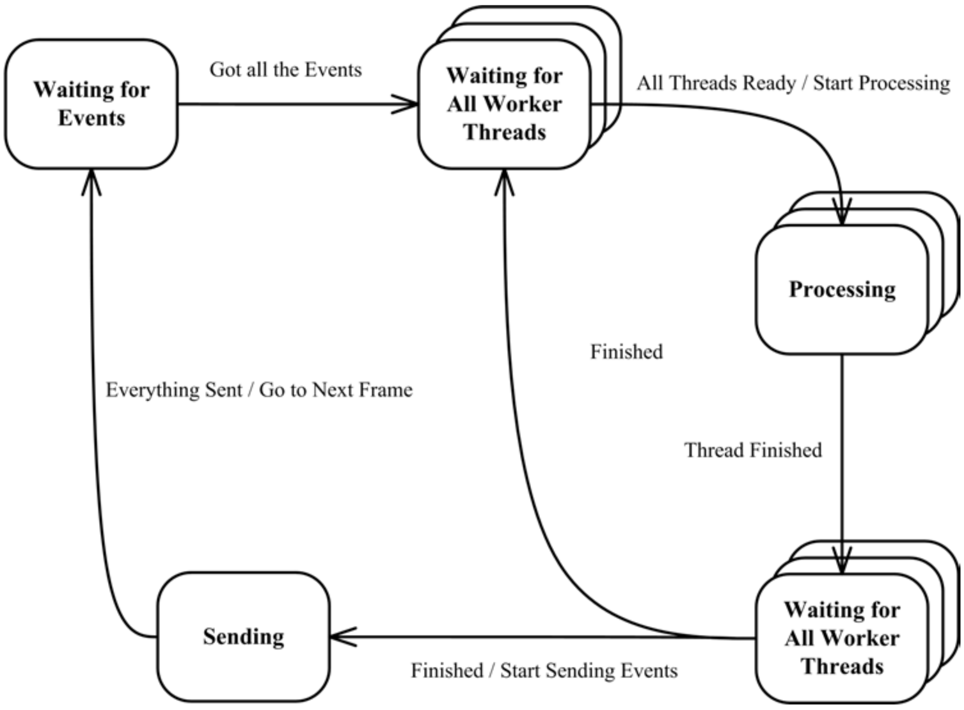

Waiting for Events
Got all the Events
Waiting for All Worker Threads
All Threads Ready / Start Processing
Processing
Finished
Everything Sent / Go to Next Frame
Thread Finished
Sending
Finished / Start Sending Events
Waiting for All Worker Threads

Figure 9.30: A Finite State Machine Showing the States of the Main Thread and Worker Threads

been adapted for this discussion. Figure 9.30 shows the Finite State Machine (FSM) for the *main* thread combined with the FSM for the *worker* threads (the layered blocks indicate that multiple threads can be in that state).

The worker threads are synchronised with the main thread by two *barriers* (see Chapter 8). This is to ensure that worker threads stop executing objects while the published issues are being sent out by the main thread. The worker threads wait at the first barrier for the main thread and the second barrier effectively forces the main thread to wait for all the worker threads to finish executing objects before continuing. Using barriers like this is very efficient, since threads use very little CPU resources when waiting on a barrier.

Having some threads do work while other threads have to wait might decrease the CPU utlisation. The backbone could be configured to have more worker threads than CPU cores. Another, probably better solution, would be to balance the load of the different worker threads in some way.

The following CSP analysis provides an alternative view to the FSM shown in Figure 9.30. There are two types of processes executing objects: the main thread and the worker threads. The main thread has to first read all the information from other nodes, synchronise with worker threads, do some processing, synch with worker threads again and then send out new information to other nodes. This can be expressed in CSP as follows:

$$MAIN \quad = \quad Read \rightarrow Sync1 \rightarrow Sync2 \rightarrow Send \rightarrow MAIN \qquad (9.9)$$

Similarly the CSP for the worker thread can be expressed as:

$$WORKER(i) \quad = \quad Sync1 \rightarrow Sync2 \rightarrow WORKER(i) \tag{9.10}$$

The threads run in parallel and synchronise on the two barriers (the *Sync1* and *Sync2* events). This can be expressed as:

$$NODE \quad = \quad MAIN \parallel WORKER(1..n) \tag{9.11}$$

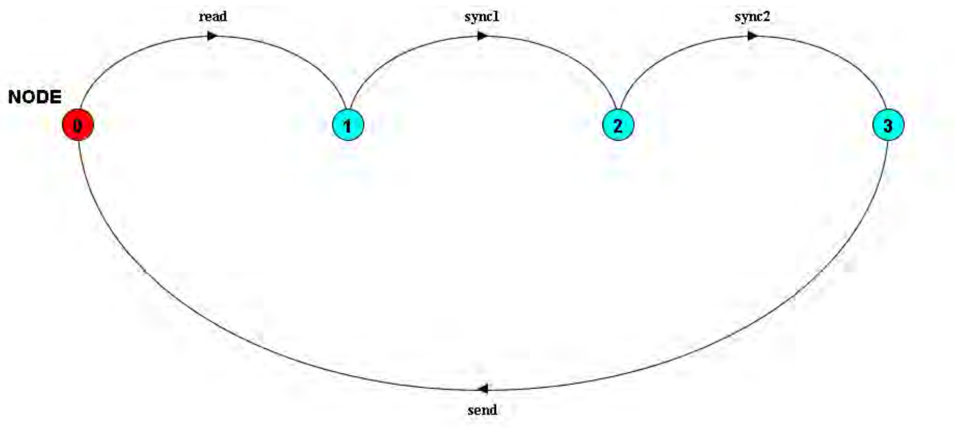


Figure 9.31: The LTSA transition diagram for *NODE*

The CSP model for *NODE* can be expressed in FSP as:

$$MAIN \quad = \quad (read- > sync1- > sync2- > send- > MAIN). \tag{9.12}$$
$$WRK1 \quad = \quad (sync1- > sync2- > WRK1). \tag{9.13}$$
$$WRK2 \quad = \quad (sync1- > sync2- > WRK2). \tag{9.14}$$
$$WRK3 \quad = \quad (sync1- > sync2- > WRK3). \tag{9.15}$$

$$||NODE \quad = \quad (MAIN \parallel WRK1 \parallel WRK2 \parallel WRK3). \tag{9.16}$$

The FSP model for *NODE* has three worker thread processes (*WRK1, WRK2 & WRK3*). All the worker threads (and the main thread) must jointly synchronise on *sync1* and *sync2* (i.e. every worker thread will wait for all the others and for the main thread to execute *sync1*, and similarly for *sync2*). Using the *LTSA* tool it was found that there is no possibility of deadlock in this model. The transition diagram generated for *NODE* by the LTSA tool is also shown in Figure 9.31.

9.4 General Discussion

This chapter discussed the framework implementation in three sections: the first evaluated the performance and scalability of applications created with the framework; the second discussed the existing C2 applications created with the framework; and, the third formally evaluated the framework implementation. This section provides some additional comments on the framework implementation.

Successful distribution of the object execution depends on how much information each model is publishing. This applies to both the parallel and distributed cases:

- When the object execution is distributed among several worker threads within one node (parallel execution) the execution speedup decreases as threads start waiting for each other. This is clear from the CPU utilisation that decreases as more objects are published. The data exchange overhead is also very small in this case since all the objects are running on one node.
- When the object execution is distributed among several nodes (on one or more hosts) the execution speedup decreases as more time is spent on exchanging data. The CPU utilisation stays high, even though the speedup decreases.

The backbone is designed for distributed object execution. Parallel object execution augments the distributed execution by utilising more of the host resources. The key to good parallel performance is CPU utilisation and solving the contention issue between the multiple threads. The key to good distributed performance is load balancing and optimising the data exchange between hosts. In general the number of titles published should be kept to a minimum.

The infrastructure and interoperability layers add to the functionality of the backbone by giving applications access to spatial simulation and interoperability capabilities. The object execution and inter object communication is however not compromised by the additional layers: each model or interoperability link is also a *backbone object*. The framework is also flexible enough to allow for discrete time and discrete event based simulation.

Each framework layer extends the capabilities of the layer below it (for example, the infrastructure layer adds capabilities to the backbone layer and the interoperability layer adds capabilities to the infrastructure layer—see Figure 8.1). Each layer does however depend on the layer that it extends. This means that the infrastructure, interoperability and simulation layers can be modified or replaced with new implementations, but the dependencies need to be managed (for example, modifying the information model defined in the infrastructure layer might affect the link implementations in the interoperability layer).

The application layer serves as a set of templates or rules that help guide developers on how to develop the relevant user interfaces and then integrate those user interfaces with the simulation and interoperability capabilities. Many of the application layer components are reusable across multiple applications. The application examples discussed in this chapter give a clear indication of the virtualisation and interoperability capabilities of the framework. The user interface and visualisation layers are application specific and do not form part of the research effort discussed in this dissertation.

Good code quality and portability is hard to measure in isolation. In general the acceptance of the framework and the way in which it is used to create applications can be taken as an indication of the code quality. Each new application developed with the framework also provides the opportunity to identify and solve potential deficiencies in the framework and to add more capabilities to the relevant layers.

The next part of this dissertation provides the lessons learned and possible future work on the framework.

CONCLUSION

The current status of the framework, the lessons learned and possible future work on the framework are discussed in this final part of the dissertation. The work discussed in this dissertation also contributes to a larger vision of unified system development within the command and control environment.