



## **A Container-Based Architecture for Simulation**

January 28, 2004

Rob Hatcherson  
Keith Holt  
Stephen Tarter

ZedaSoft, Inc.  
2310 Gravel Dr.  
Fort Worth TX 76118  
817-616-1000  
<http://www.zedasoft.com>

© Copyright 2004 by ZedaSoft, Inc.

All rights reserved. All trademarks are property of their respective companies.

## Introduction

ZedaSoft's Container-Based Architecture (CBA) is a full-featured and easily extensible object-oriented simulation framework. Development of CBA started as an effort to redesign Lockheed Martin's F/A-22 marketing and concept demonstration simulators, but ended up being much more. This paper is a high-level discussion of our design motivations and the resulting architecture.

## History

CBA was first conceived in the summer of 2002 by the ZedaSoft design team, whose members included the authors of this paper. At that time each member of the team had worked in the flight simulation industry for 20 years or more, and each had extensive experience with object-oriented development platforms outside of simulation. We looked at the design and development of CBA as an opportunity to combine our experiences from both of these worlds into a better simulation framework.

Prior to starting on the design we spent some time thinking about the architectural approaches we had seen during our careers. One feature that stood out was that all simulations we had worked on were "ownership-centric". To build a new simulator somebody rolled a cockpit shell complete with avionics displays into a room, and told us to make it go. All subsequent software development work revolved around the cockpit. The cockpit was the simulator, and the software was a peripheral resource that catered to the cockpit's needs.

After a little thought we concluded that this approach was backwards and restrictive; the cockpit should be nothing more than a station where displays can be realized and one or more human operators can interact with a virtual entity. The "ownership" might be the most sophisticated entity in a simulation, but otherwise should not enjoy any more privileges than the entities around it. We should be able to roll a physical or virtual cockpit up to a simulation, plug in to the network, and interact with any compatible entity.

Eliminating ownership-centric thinking in favor of a peer-based approach opened the door to a number of additional possibilities that became core design goals.

## Object-Oriented Terminology

There are a lot of concepts in object-oriented design, and we frequently mention a few of them in this paper. For the uninitiated reader we'll establish some simple definitions:

*Class* - a collection of data and executable code that together defines a model of a concept or real-world thing. Classes can inherit data and executable code from other classes. A class from which other classes inherit is called a "superclass" of those classes. Classes that inherit from another class are said to be "subclasses" of that class. Classes can be aggregated hierarchically to build up complex models. Classes usually must be "instantiated" to produce executable things, although we often define executable operations that can be requested of the class itself.

*Instance* - when we talk about "objects" we're really talking about instances of classes. A class can be thought of as a template for things that might be created, and instances can be thought of as the created things. Zero or more instances of a given class can be created, provided the class doesn't suppress creation of instances. This process is called "instantiation".

*Method* - methods (also called "member functions" depending on who you're talking to) are executable operations that can be performed against either a class or an instance.

*Interface* - an interface defines a set of methods without defining their implementation. Interfaces can be implemented by zero or more classes. Interfaces allow us to define "loosely coupled" systems, i.e. systems that minimize explicit dependencies between classes.

## Design Goals

Although the overall requirement of the initial CBA effort was to end up with a new-and-improved F/A-22 simulator, we decided to open up our thinking and set general design goals that could be used to realize a simulation for any group of arbitrary entities:

- The design must be object-oriented from the ground up.

The simulation problem domain is a natural fit for object-oriented design at almost every level of detail. At the highest level the simulation is a collection of entities including (but not limited to) aircraft, ships, and land vehicles, all operating together in some virtual environment. Each entity is another, finer-grained, aggregation. For example, an aircraft is composed of an airframe, engines, fuel system, onboard systems, a pilot, and whatever else is needed to complete the model. Some of these systems may be further subdivided, such as modeling the individual tanks in a fuel system.

- The design must be loosely coupled.

The design must have no class dependencies that might defeat flexibility. Interfaces should be used wherever possible as a basis for the interaction between objects.

- The design must be extensible without requiring modification of the core system.

We must be able to introduce new types of entities without having to make changes to the core system. Loosely coupled classes were key to supporting this goal.

- The design must eliminate artificial limits on scale.

The legacy simulations that our team previously was involved with typically used arrays to model sets of entities. A side effect of this approach was that the maximum number of entities that could be modeled was defined at the time the system was built. For example, there might be a limit of 20 air threats, or 5 bombs in flight simultaneously. This approach introduced artificial limits on scale that made the simulation difficult to extend. By "artificial" we mean that even though the simulation host might have had the potential to support more, the software's internal limitations prevented that processing potential from being used. The design must be such that the processing potential of the simulation host computer and network were the only barriers to scale.

- The design must support a multi-viewer capability.

When we abandoned ownership-centric thinking and started thinking of the cockpit (or more generally, the operator station) as nothing more than a display and interaction point for an entity, we opened up the possibility for zero or more stations to be in operation simultaneously for a given entity. These capabilities provided a built-in manned control station capability, as well as an operator station repeater capability.

- The design must promote cross-platform executable component reuse.

A great deal of time and effort has been spent in the simulation community inventing ways to exchange data between simulation silos, DIS and HLA being two obvious examples. We wanted a system that supported executable component reuse in addition to data reuse. It should be possible to maintain a depot of ready-to-use classes that can be downloaded and instantiated at the time a simulation scenario is realized.

## So What's A "Container"?

At a recent trade show two guests walked up to our booth, and one of them read aloud our sign that said "Container-Based Architecture". He paused for a moment, then turned to his associate and said "speak English".

The term "container" is borrowed from the Java world. A container provides an execution environment for components that cannot execute standalone. Another way of thinking about it is that a container is a receptacle for plug-ins. An example from Java is the servlet container, which provides an execution context for components that generate dynamic web site content.

Containers interact with their contained components through predefined interfaces or superclasses. In an object-oriented sense these interfaces may be realized by inheriting from some particular superclass, and/or by implementing one or more interfaces.

During our design process we quickly realized that this model translated perfectly to the simulation problem domain. Why couldn't the simulation be modeled as a collection of plug-in components executing under control of a container?

## Container Design

Thinking about the simulation problem domain in terms of a container and contained objects led to the following definitions:

*Simulation Container* - an execution context for instances of plug-in components that model concepts from the problem domain. The container provides life-cycle management for its contained participants, provides access to a virtual environment in which the participants play, and provides general utility services that allow the participants to interact with each other.

*Simulation Participant* - an object that executes within a simulation container, and interacts with the container through a life-cycle interface.

*Entity* - a type of simulation participant that serves as a superclass for models of real-world things such as aircraft, sea vessels, and land vehicles. Entity objects:

- Come and go during a container's lifetime (for example, when a missile is fired it becomes an autonomous entity in the container);
- Exist at a point in space, and move about in the environment;
- Maintain state information that is regularly published on an external data bus.

*Environment* - a description of the virtual world within which entities execute. The environment is independent of the entities participating in the simulation. For the simulation problem domain the environment may include a description of the earth model in use, the characteristics of the atmosphere, the layout of the terrain, a model of the sea state, and anything else an entity may need to know to perform its updates. Like an entity, the environment is a dynamic object that may change over time. For example, the characteristics of the atmosphere may change as a function of time of day, or the terrain description may change due to a munition detonation.

Environment objects:

- Live as long as the container, and cannot be destroyed or removed (though they possibly may be modified or replaced);
- Typically do not exist at only one point in space, and do not move (though they may regularly update their properties);
- Do not regularly publish state data over an external data bus, but instead acting as services that provide data to entities upon request;
- May provide network notification of events; for example, a terrain description may provide a notification that it has been modified due to a munition detonation.

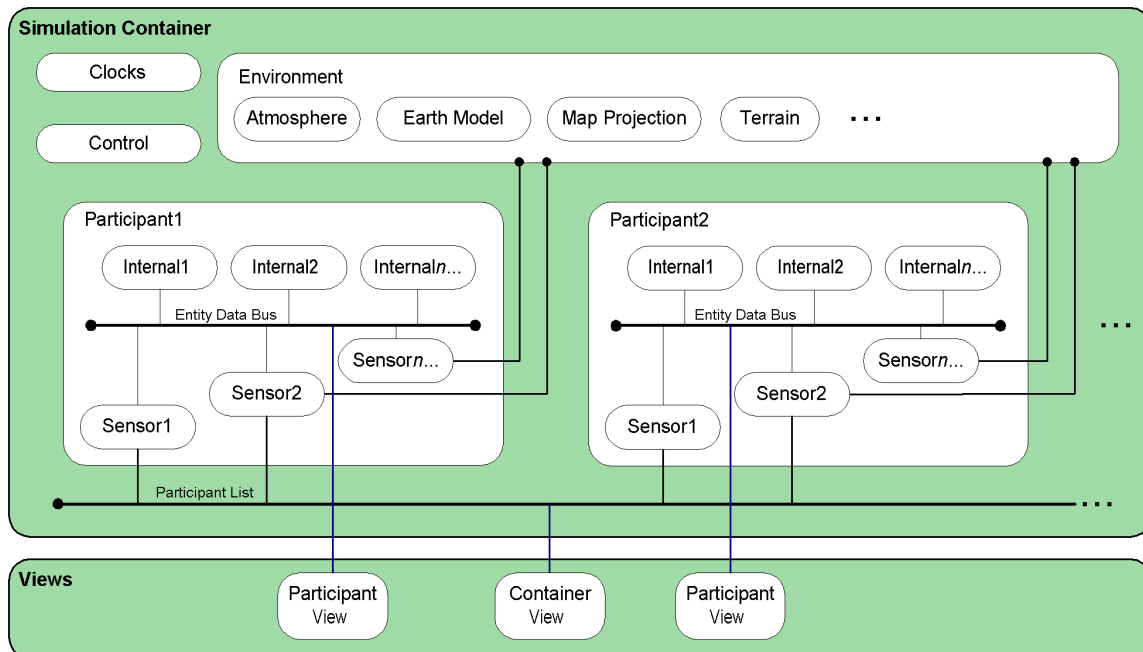
*Non-Entity* - an object that is not an entity but is still a simulation participant. These objects operate in the container just like entities do, but do not represent real-world things. Examples of non-entities are controllers for external devices such as image and audio generators, and scripting engines for scenario control.

*View* - a way to peek inside a simulation container. Simulation containers and their contained participants do not themselves provide any user interfaces; this job is left up to views. Views such as a virtual cockpit may provide a look at the interfaces of a single entity, while views such as a mission overview may provide a look at an entire simulation container.

*Data Bus* - a channel through which logical intra-entity data is shared. For example, container-side data sources communicate with their views over a data bus.

*Plug-In* - the extension mechanism that allows new types of simulation participants to be added to the system. A plug-in defines both the simulation participant component that executes within the container, as well as any views that can be used to examine and/or interact with the participant. Because simulation participant and view components are defined in terms of interfaces, the core system can interact with them without having to know anything about their implementation.

At this point you can start to envision what a system might look like for simulation:



The container provides a number of services to its contained participants:

*Top-Level Execution Control* - the container controls the main run loop, and therefore serves as an "executive" for the simulation.

*Life-Cycle Management* - the container manages the execution of each participant through a life-cycle interface. This interface defines methods for, among other things:

- Notifying a participant that it's about to be added or removed from the container;
- Notifying a participant that other participants are about to be added or removed from the container;
- Telling a participant when to perform its routine updates;
- Notifying a participant that the container is about to terminate.

*List-Based Processing* - the container maintains the participants in an open-ended list, making the processing power of the simulation host and the capacity of the network the only barriers to scale. Each participant has write-protected access to the participant list and can access public information about any other participant. For example, a sensor model on a vehicle may look through the participant list to determine which other participants it can "see".

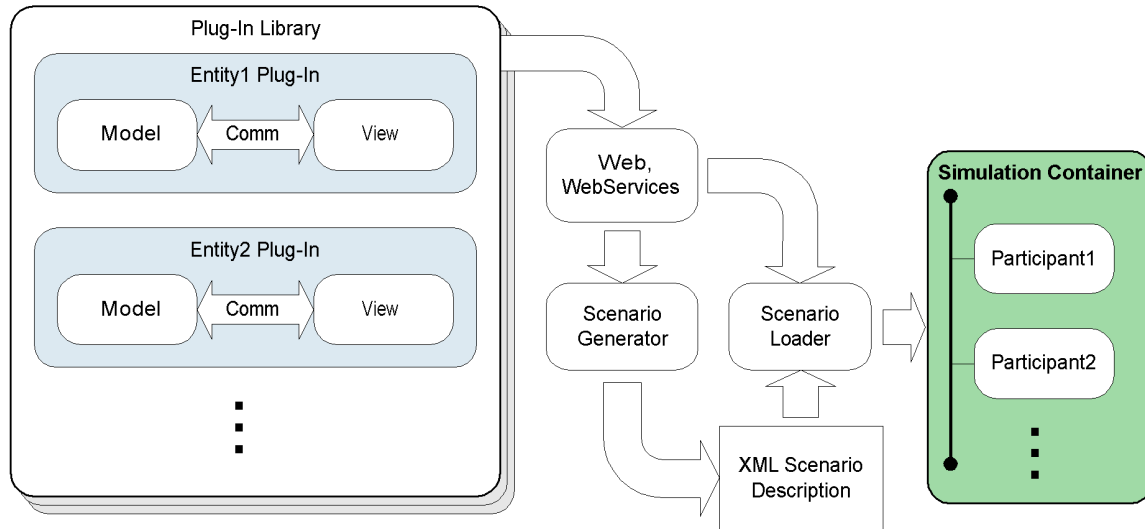
*Categorization* - the container can maintain subsets of the main participant list based on some categorization criteria such as class membership or property value evaluation. These categories are updated as participants come and go in the container. For example, a simple radar system may only care about entities that happen to be air vehicles, and a category can be set up to maintain a subset of the main participant list that meets this criterion. If another entity comes along later and asks for a category with the same criteria, it will be provided access to the same category. This prevents each participant from having to maintain possibly duplicated subset lists.

*Publish/Subscribe Events* - the container maintains a publish/subscribe event center where an arbitrary number of participants can register to be notified when events of interest take place. This permits the originator of the event to not have to care about which participants are interested, and leaves it up to the recipient to decide how the event is handled. An example is munition detonation. When a munition detonates it posts a notification to the event center. The event center then notifies interested parties, who make a decision about how the event affects them. An image generation controller may take the opportunity to create an explosion in the visual scene, an audio controller may create an audible explosion, and participants that represent real-world objects may perform damage assessment. This mechanism contributes to the extensibility of the system by keeping the object that posts the notification from having to know what other objects to send the notification to. If new types of objects that care about a particular type of notification are added to the system they simply register to be notified, and nothing else has to change.

*Execution Sandboxes* - simulation participant operations are always performed in a sandbox that protects other participants from error conditions. If a participant encounters a fatal error, it's simply removed from the container.

## Container Creation

CBA provides a framework through which platform-independent plug-ins can be located and loaded over local or remote networks. The information provided by the plug-ins is used to create scenario descriptions, and to build executable containers.



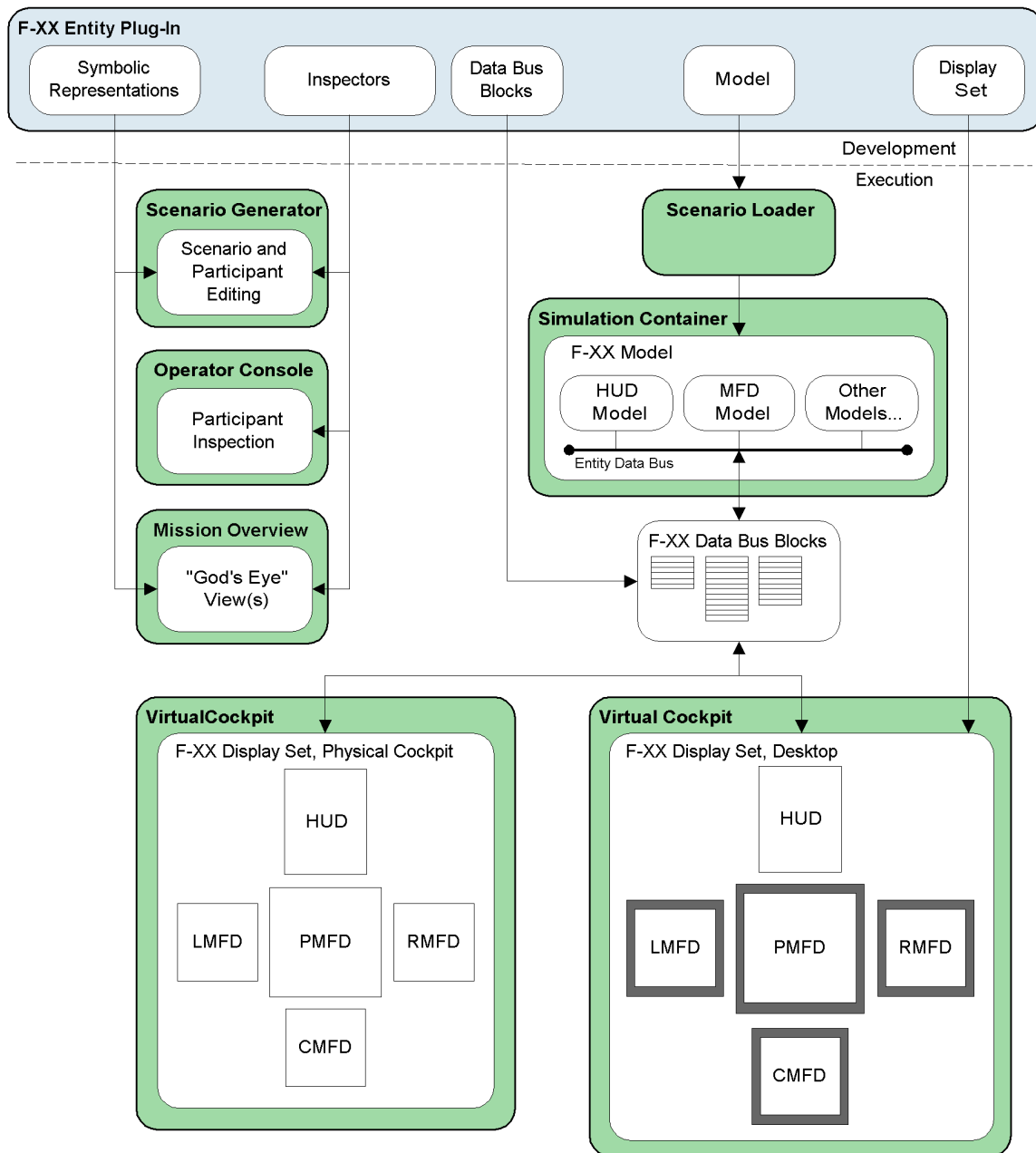
The ScenarioGenerator allows the user to interact with entity plug-in information to create scenario descriptions. Scenario descriptions specify what instances need to be created to build a container, and what values should be used to initialize those instances. Scenario descriptions are stored in plain-text XML files.

The ScenarioLoader is responsible for creation and execution of simulation containers. The loader consumes an XML scenario description file, locates the required classes either locally or over the network, creates instances as specified in the scenario description, and through a runtime interrogation mechanism determines information about the involved classes and establishes any specified initial values. The loader then starts the container's main execution loop, at which point remote clients such as views are free to connect to the container and/or its participants and interact with them.

The ScenarioLoader is capable of interacting with any participant classes that follow a few simple implementation guidelines, even if those classes were unknown to the loader prior to encountering a reference to them in the scenario description.

## Container Extensibility

New types of participants are added to the container by implementing a plug-in. The major components of the plug-in are the container-side components that implement the participant's model, the view-side components that implement the participant's interfaces (if any), and the data components that go between them. Both the container-side and view-side components are defined in terms of interfaces, so the core container and view systems can interact with the new components as long as the interfaces are faithfully implemented. The communication channel between the container-side and view-side components can be anything as long as the endpoints agree on mechanism and it's well behaved. The following illustration shows the overall structure of a hypothetical F-XX fighter plug-in, and how it is distributed for execution:



A key goal of using the plug-in approach was to avoid another problem that existed in legacy simulations we had worked with. In those systems it was common for the introduction of a new type of participant to cause a ripple of minor changes across the entire simulation. For example, introduction of a new type of aircraft may cause a mission overview to need a new kind of graphical representation for that aircraft, and cause a scenario generation system to need a new set of inspection panels for that aircraft's attributes. We thought there was something fundamentally wrong with needing to disturb stable, fielded applications just to expand the kinds of things that can participate in the simulation. The plug-in approach solved this problem by predefining the interfaces through which applications could interact with participants, and defining applications that needed to interact with those participants in terms of these generalized interfaces. In this way new types of participants could be added without disturbing the applications as long as the participants implemented the interfaces properly.

In CBA there is no inherent limit to the number of plug-ins that the system can simultaneously support; the only practical limit is the available resources of the simulation host. Furthermore, plug-ins can be loaded from a variety of locations. For example, plug-ins can be loaded from the local file system, from another machine on the local network, or even over the web, provided accessible network paths are provided.

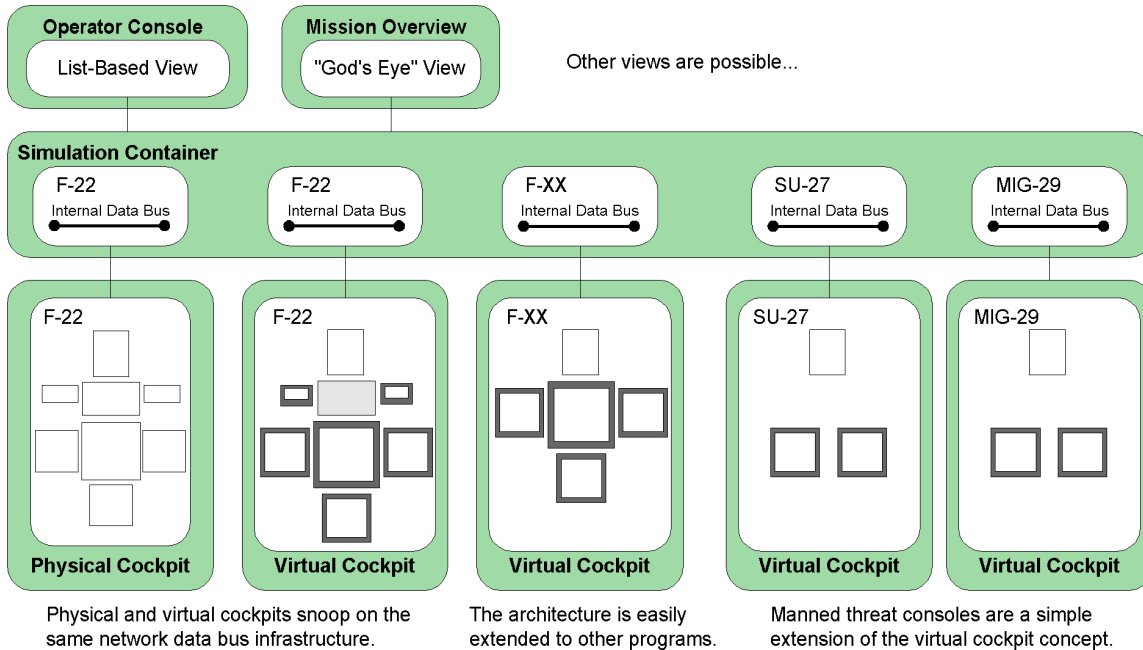
A running container can load an accessible plug-in at any time. This means that participants of any kind can be brought to life in a container both at start up, and while the container is running.

## Views

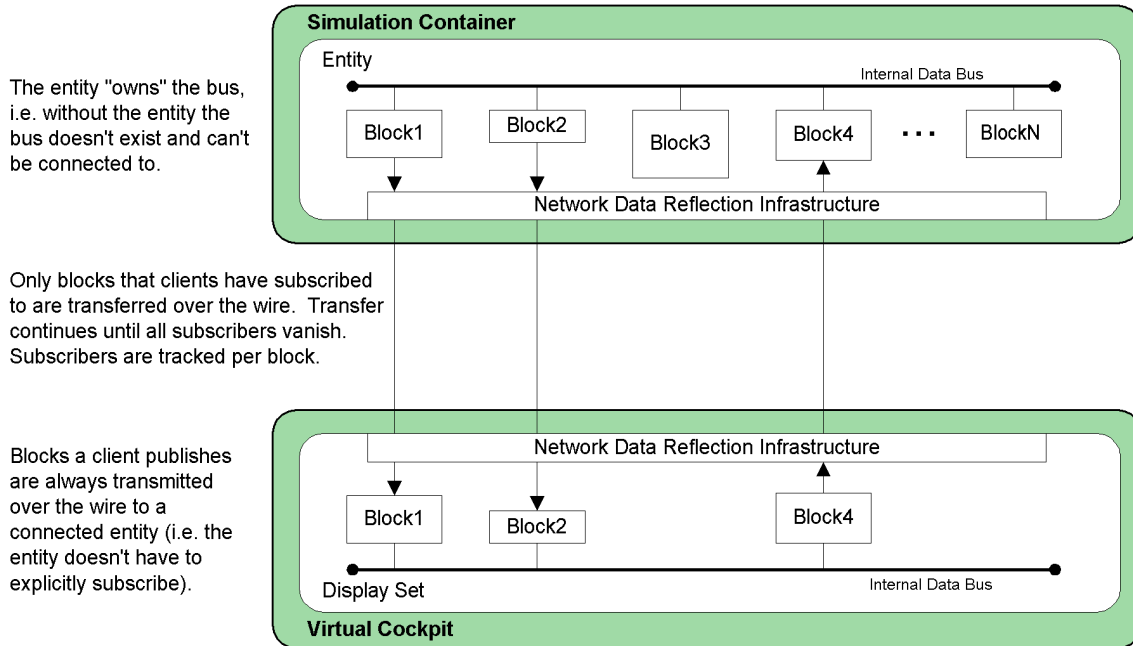
The simulation container is a data engine, and doesn't perform any graphics tasks. This job is left up to views that know how to interface with peer objects on the container side. A view may show information about the entire container, a single participant within the container, or anything in between. The initial design of CBA included three main view applications:

- The VirtualCockpit view is itself a container for the display set of a single entity (the term "virtual cockpit" is a remnant of the flight simulation domain; this application might be better called the VirtualOperatorStation). Each entity plug-in can provide an optional display set that defines the graphical presentation of its operator control views. For example, an aircraft entity may provide views of the various cockpit displays the pilot can interact with, including virtual bezels and panels. When the VirtualCockpit starts it discovers running simulations via a naming service, and allows the user to select an entity of interest from a particular simulation. The entity's operator control display set is then dynamically located and brought to life on the VirtualCockpit user's station.
- The OperatorConsole is a multi-entity view that provides a list-based look at all participants in the simulation. Participants can provide editing and/or inspection views that can be accessed from this application. The user can also monitor and control the simulation container's state from this view.
- The MissionOverview is a multi-entity view that provides a simple God's-eye plan view of all simulation participants that represent real-world objects. This view effectively provides what the OperatorConsole provides, except that a subset of the total participant set is shown, and a graphical view is used in place of a list.

The only restrictions on the number of views that can simultaneously interact with a given container are available network bandwidth, and the ability of the participants in the container to prepare, send, and receive data. Multiple VirtualCockpits can be used simultaneously, providing a built-in manned control station capability, and can also simultaneously connect to the same entity, providing a built-in display repeater capability.



In the current version of CBA views communicate with their container data sources via what is essentially a network-based reflective memory system. When a remote view starts it negotiates a communication channel over which data bus data can be sent. This channel is used to support a distributed publish-subscribe system where data blocks on the bus can be requested by remote views. From the view's perspective the data exists locally as if the view were an intrinsic part of its peer in the container. This mechanism is what allows the view and its container-side peer, usually viewed as a single logical entity, to be physically distributed across the network.



Note that this is just one possible communication channel. Views and their container-side peers can use whatever communication technique they want, as long as it is well behaved.

## Summary

CBA's ability to assemble arbitrary component sets together allows the core system to be extended without modification, and the result is an extensible development and execution framework that can be applied to many simulation domains. CBA's network-centric approach to data sharing, peer approach to entity modeling, and list-based processing provide levels of flexibility that traditional simulation designs don't support.

CBA was implemented using Sun Microsystems' Java development platform. This flexible and productive platform allowed us to implement the most powerful features of our design in a way that could only be accomplished with difficulty in C++. CBA's ability to load and execute centrally located class files (some of which it may never have seen before) across a variety of platforms, without modification, is a direct result of having used Java. These features also permit end-users to easily extend CBA with their own classes.

The Java implementation of the CBA design is not yet suitable for hard real-time applications due to restrictions in current Java platforms, so for the near-term use must be restricted to soft real time or batch simulations. However, two industry groups are working on the real-time Java problem, and have made great progress. The issues around why we chose the Java platform, along with more information on the work in progress to support real-time Java, are discussed in the ZedaSoft white paper titled "Using Java For Soft Real-Time Simulation".