Simulation Programming Guide

Draft: 20170217

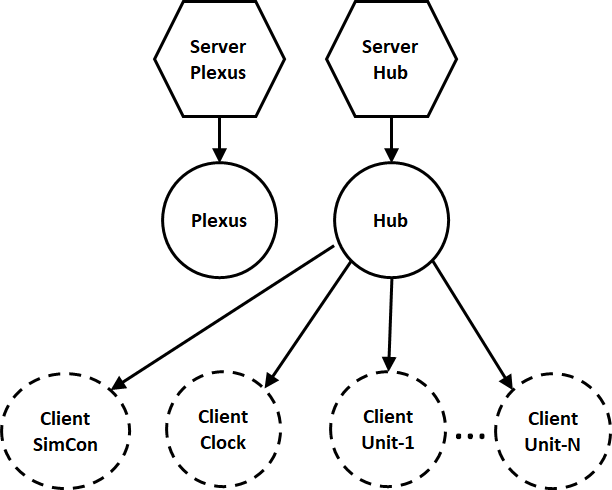
# Introduction

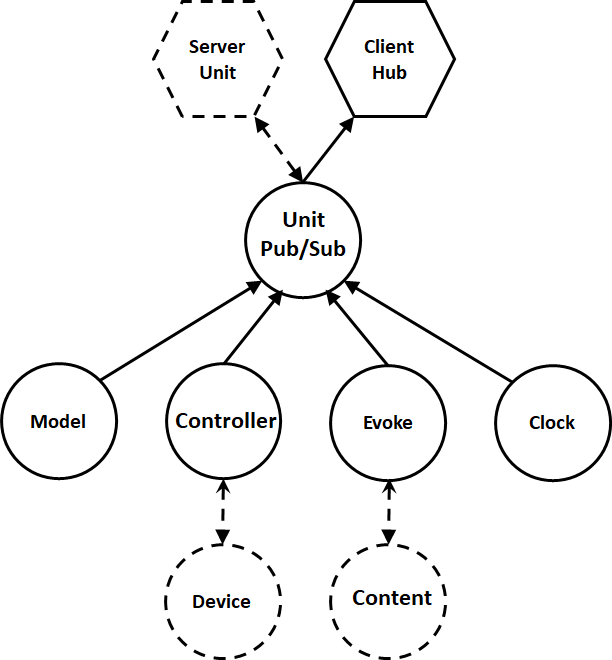
The purpose of this document is to provide a basic understanding of the core capabilities of the xSim package of modules and systems. It is meant to be completely generic in that it provides services that can be used in any simulation but none for any specific simulation. The core concept here is that a simulation comprises systems of systems. The top-level system is referred to as the ‘Hub’, and each of the subsystems are referred to as ‘Units’.

On disk, the directory structure is divided into several key sub folders. The ‘Core’ folder contains those modules that are common to many different systems while those modules in the ‘Units’ folder are specific to the implementation of a specific capability. Note that in a module name might be duplicated across units, reflecting the particular function of that module. For example, an ‘Evoke’ module performs differently for each unit, possibly creating a different Popup widget on a display, while at the same time it is good practice to know that an ‘Evoke’ module performs much the same function in each unit. Of course, this is not enforced in any way, and developers are completely free to create chaos in any manner that they choose. The ‘Widgets’ folder (to be renamed ‘Web’) contains those modules that are only present in a browser system. The ‘Systems’ folder services its usual purpose of organizing the various systems used in the development of the xSim module. The systems that support particular vertical application will not be included here, but will be present in repositories or brokers that support those vertical. For convenience in testing we are currently including some DOE modules here, but they will be removed before release.

The ‘Systems’ folder is organized somewhat differently in that each system has any number of subfolders, each corresponding to a system that is included in the current system of systems. Initially each system folder and subfolder need only contain a config.json file. When the system is started up, execution starts in the top folder which contains the ‘Hub’ implementation. The ‘Hub’ then scans its working space and spawns each of the ‘Unit’ subsystems. This approach is somewhat limiting, as in general a simulation is expected to be run on multiple CPUs, so this is just a convenience during development. In the general sense, a subsystem can be initiated on a remote computer and linked into the system through the Plexus module by sending a ‘Join’ command to the ‘Hub’ module in the ‘Hub’ system. For purposes of initial development, the name of the subsystem folders is also the Plexus channel used for communications.

# Core Architecture

The xSim architectural framework is somewhat analogous to the relationships between Proxy modules and a single Plexus. In the case of xSim, the singleton is the ‘Hub’ systems. Each of the subsystems contains a ‘Unit’ module that is analogous to the Proxy. The ‘Hub’ and all of the ‘Unit’ systems have a two-way communications link implemented as a pair of Server/Proxy pairs. Each system of systems has one ‘Hub’ system and N ‘Unit’ systems as shown in the accompanying figure. The solid glyphs are modules that are instantiated at startup for the ‘Hub’ system as they are elements of the core config.json. At startup, the ‘Hub’ modules scans its working folder and spawns all subfolders that contain a config.json file as a ‘Unit’. As the ‘Unit’ subsystem starts up, it sends a ‘Join’ message to the ‘Hub’, at which point the ‘Hub’ generates a new client Proxy to the joining unit. This mechanism also supports the idea that any system can ‘Join’ a system of systems simply by sending a ‘Join’ message to the appropriate ‘Hub’ system. The Proxy modules shown in the diagram with dashed lines are those created in response to a ‘Join’. They would be eliminated if a system left the system, but this functionality has not at this point been implemented.

The ‘Hub’ system shares functionality with what in other architectures might be a separate ‘Plexus’ system. Since any system has at most one ‘Plexus’ and one ‘Hub’ this is a natural approach to eliminate a certain amount of complexity. A separate ‘Plexus’ systems could also be used, but there is seemly no advantage in doing that. Suffice to say that it is not required that the ‘Hub’ and ‘Plexus’ functionalities coexist in the same system.

A diagram showing the functionality of a typical unit is shown on the left. Every unit system has one ‘Unit’ module, as the ‘Hub’ and ‘Unit’ are paired in much the same way as a ‘Plexus’ with ‘Proxy’ modules. When a unit starts up and before it sends the ‘Join’ to the ‘Hub’ it instantiates a ‘Unit’ specific server which is shown as dashed since it is not original created from the config.json. The ‘Unit’ module is vanilla and should not be further specialized as it is used in many architectures beyond the simulation one being discussed here. In addition to providing the initial connectivity with the ‘Hub’, it also provides a Publish/Subscribe service. The third tier of modules shown represent possible modules that provide the actual functionality of a subsystem. The only one that is required is the one labelled ‘Controller’ which provides the core functionality. In many applications this is the only module required, although it is just as likely that functionality can be distributed in any number of collaborating modules which of course can communicate with each other in any manner desired. All of the third tier of nodes are connected to the ‘Unit’ module using the publish/subscribe mechanism. For example, the module labeled ‘Module’ subscribes to the ‘GetModel’ command and when that command is received by the ‘Unit’ it is forwarded to the ‘Model’ module for processing. The tier three modules can use this method for communicating if desired for the sake of interchangeability of parts, but messages can also be received from external sources through the ‘Unit/Server’.

In the xSim repository a ‘Seed’ unit is provided in the ‘Units’ folder that can be used as a starting point for new modules. This module has only the basic functionality can be copied and specialized in any manner desired.

For simulation the other three modules are examples of the functionality that might be related for a specific application, for this case, simulation. For other uses, such as earthquake data processing, the ‘Controller’ would still be present but other auxiliary modules would be present in a domain specific manner. In the simulation application, specifically related to IoT simulations, the other three modules, ‘Model’, ‘Evoke’, and ‘Clock’ provide functionality that is specific to simulations. The first two are strongly related to the method of visualization being used, although many visualizers could in principal share these modules. The ‘Model’ delivers a module that can be used in a 3D visualization, while the ‘Evoke’ module delivers content that can be visualized in a popup. In the ‘Seed’ the model is an ‘X3D’ formatted model, but this is not required and only need be consistent with the visualized being used. For simulations this is the ‘EVE’ (Existential Visualization Engine) visualizer which is just another ‘Unit’ can ‘Join’ the simulation while it is running. It is referred to as existential as it is not necessarily the only visualizer imagined and it can join a simulation that is already running. The architecture supports multiple instances of visualizers so that development can be collaborative if desired.

For IoT or Microgrid modeling, an additional module is shown. The concept here is that the ‘Controller’ is the brain and the ‘Device’ is a representation of the physical device. For example, if the device is a battery, the ‘Device’ component simply provides what would be called the ‘Plant’ model for a battery, or generator, and all of the control logic is present in the ‘Controller’ module. Best practice is to use the same protocol between the ‘Controller’ and the ‘Device’ as the physical device would use. In that manner the simulation device can be replaced by a physical device (HIL or Hardware in the Loop). Specifically, if the device is controlled using MQTT protocols, then the simulation should at least use messages that have the same content if not the actual string representation of the MQTT commands to communicate with the device both for control as well as sensing.

Similar to the ‘Model’ module, the ‘Evoke’ module subscribes to the ‘Evoke’ command and responds appropriate when the ‘Evoke’ command is sent from the EVE visualization layer to the simulation. While the ‘Model’ module is more general, and an X3D representation could be used in many visualizers including VR, the ‘Evoke’ response is specific to the ‘EVE’ visualization component running on a browser. For ‘EveView3D’ Module is the browser component that paired with the EVE server unit.

The ’Evoke’ module can have an ephemeral child shown as ‘Content’ in the diagram. The purpose of that is to allow configuration of Unit specific content (e.g. Dashboard) in a canonocial popup widget. Three base vanilla popup base modules are provided, PopHTML, Pop2D using PIXI.js, and Pop3D using Three.js. These three vanilla popups interact with the implementation code in Content.js to provide the Unit specific UX. An extremely simple example of the Content.js implementation is as follows:

(function Content() {

//-----------------------------------------------------dispatch

var dispatch = {

SetPar: SetPar,

Activate: Activate,

Cleanup: Cleanup

};

return {

dispatch: dispatch

};

// This is called by Evoke on the server to allow the content

// creator to add parameters such as window size to the

// parameters that are to appear in the Par of the widget

// maker on the browser.

function SetPar(par) {

par.Height = 100;

par.Width = 200;

}

function Activate() {

console.log('SeedHTML/Content/Activate!');

// console.log(JSON.stringify(this.Par, null, 2));

var Vlt = this.Vlt;

var Par = this.Par;

var that = this;

Vlt.text = $(document.createElement('p'));

Vlt.text.html('Hello Views!');

Vlt.idTimer = setInterval(function() {

console.log('Activate', Par.Controller, Par.Name);

var q = {};

q.Cmd = 'GetData';

q.Name = Par.Name;

// log.w('stop', JSON.stringify(q));

that.send(q, Par.Controller, function(err, r) {

console.log('Data:', r.Data);

});

}, 1000);

}

function Cleanup() {

console.log('--Content/Cleanup');

var Vlt = this.Vlt;

clearInterval(Vlt.idTimer);

}

})();

This is a self executing function returning a dispatch table which is used by the popup base class to communicate with the content code. The ‘SetPar’ function is actually used on the server side to set parameters that will be available to the content code on the browser. This is used fo set things lice CSS parameters or window size as shown here. The ‘Activate’ function is executed on return from the ‘Evoke’ command sent to the specific Unit. This is where the initial rendering occurs. Finally, the ‘Cleanup’ function is called when the close box is clicked to allow for the content code to do such things as destroy timers and such, as shown in the code example. Notice that the example shows a periodic query sent to the associated unit with its response. Any commands can be used, as long as they are registered and processed by some modules in the associated Unit.

# Simulation Specific Structures and Modules

There really only two modules that are specific to simulations. These are the ‘SimCon’ model in the ‘Hub’ system, and the ‘Clock’ module that is present in both the ‘Hub’ as well as every ‘Unit’ system that is integrated into a simulation. The purpose of the ‘Clock’ model is to provide the sense of time necessary for simulations in a manner that the simulation can be accelerated so that a developer does not have to wait for a year to find out how a system might respond to a year’s worth of data. It actually comes in two flavors. In the ‘Hub’ the clock is a master unit synchronizes all of the slave clocks in the ‘Unit’ systems. What determines whether it is a slave or master is the presence of several parameters in its ‘Par’. Specifically, a master clock is provided with a start time, and stop time, and an acceleration scale. It also has a synchronization interval in seconds that determines how frequently in wall time that it synchronizes the unit slaves. The master clock dispatches a ‘TimeSynch’ message through the ‘Hub’ server, which sends it to each ‘Unit’. On the ‘Unit’, the slave clock subscribes to the ‘TimeSynch’ message and as such receives and processes these messages as they are received.

The other simulation specific module is the ‘SimCon’ which is currently under development. Its purpose is to start and stop the simulation, to cycle the simulation through multiple simulations over the same time frame with varying environmental data for example, and to trigger reporting when a designated number of simulations have been completed. This paragraph will be updated as soon as it is determined exactly how this module functions.

# EVE: Existential Visualization Engine

The EVE is for the purposes of simulations a manipulation tool which is essentially outside the system that it is manipulating, which why it is called existential. It can join and leave running systems, and more than one EVE can view and control a system at the same time. Since both EVEs are visible as Icons to each other, it would be a simple task to allow a chat window as a popup.

There is an important lesson to be learned in the implementation of the EVE. Basically it queries the ‘Hub’ with a ‘GetUnits’ command, which returns a list of the channels to all of the individual units. This allows EVE to talk to any unit individually, or it can also broadcast a message to all connected units as well.

This section will be fully developed at a later time.

# SimGate: Gateway Simulation Unit

There is no clearly defined Gateway in the sense of the DOE Phase II contract, so this one should be thought of as one of a pantheon of possible Gateways that serve special purposes. Also, there is some disagreement on just how a Gateway should interact with the simulation. This particular implementation does three things: 1) pass through and modify pricing signals, 2) start, stop, and set modality of the simulation, and 3) collect results for later analysis. The modalities include real-time, accelerated time, and stepped time. A peculiarity of the implementation is that the Gateway does not need to be part of Units subordinate to a single Hub. The implication here is that different Gateways can be used for the same collection of Units without duplicating the Units in another directory tree. It is even possible for two Gateways to share the same Units at the same time, as long as the Units are not designed to save state.

# Vision for Future

The framework discussed here is intended to be much more general than what is required for simulations. The essential components are ‘Hud’, ‘Unit’, and some implementation of a ‘Controller’ in each unit. The core architecture supports all three of the domains identified in the ‘What is xGraph’ white paper. For example, for the StarCraft II implementation, the various units and buildings would be represented by separate systems, and the ‘Device’ would represent that capabilities of a specific unit. The controller would be split into multiple modules representing such roles as harvesting, exploring, or attacking. It is also submitted that the systems interoperability architecture described here is probably a good starting point for the architecture that is manipulated by the capability referred to as ‘xEvolved’ or ‘xEvolved Pro’. In this case, the module pair EVE and EveView3D would be replaced with their xEvolved counterparts.