ACM
INTERNATIONAL COLLEGIATE
PROGRAMMING CONTEST

California State University, Sacramento's

PC^2

Version 9

Technical Specifications

<Last Update: 2019-05-22>

Table of Contents

1 Introduction

PC^2 is a dynamic, distributed real-time system designed to manage and control Programming Contests. It includes support for multi-site contests, heterogeneous platform operations including mixed Windows and Unix in a single contest, and dynamic real-time updates of contest status and standings to all sites. It provides support for teams to submit *runs* (programs intended to solve contest problems) to Judges, and provides support for using either or both of two distinct methods of judging: *automated* (or *computer*) judging and *manual* (or *human*) judging. The system also provides a variety of mechanisms for displaying current contest standings (*scoreboards*), as well as support for allowing external systems to monitor the state of the contest via an *event feed* facility and a variety of web-based information access points.

This document describes the internal structure and operation of the PC^2 Version 9 ("PC2v9") system. PC2v9 is a collection of Java programs, along with a few additions such as some control scripts, some XSL files used to control scoreboard displays, and some PHP code supporting certain web-based operations, along with a full set of both user- and system-level documentation. The system is organized as an Eclipse Java Project and is housed in a Git repository on machines hosted by the College of Engineering and Computer Science (ECS) at California State University, Sacramento (CSUS).

The top level of the PC2v9 Eclipse project contains the following folders:

Folder	Contents
`src`	Java source code for the various system components
`test, testdata, and testout`	a variety of Java JUnits for regression testing, along with the data used by those tests and the output produces by running the tests
`bin`	scripts used for module startup and various administrative operations
`data`	XSL files for generating formatted scoreboards
`dist`	current distribution versions of various modules
`doc`	PDF and MS Word (source) versions of various system documents
`images`	image files used by various modules
`projects`	system components which were organized for historical reasons as separate projects (but which are now integrated into the PC2v9 system),
`samps`	a variety of useful files including sample contest problem solutions in a variety of programming languages as well as a number of useful scripts for performing various contest-related operations.
`vendor`	a collection of third-party libraries used by the system

Additional folders created in the project hierarchy when the system is executed within Eclipse include **`logs`** holding various modules logs and **`profiles`** holding configuration profile information.

This document assumes the reader is familiar with the overall operation of a programming contest, and that the reader is also familiar with the user-level steps involved in installing and configuring a PC^2 system. For further information on PC^2, including how to configure the parameters which control and enable various judging operations, refer to the *PC^2 Contest Administrator's Installation and Configuration Guide,* which can be found at http://pc2.ecs.csus.edu/. Additional information about PC2v9 can also be found on the PC^2 Wiki at http://pc2.ecs.csus.edu/wiki/Version_9.

2 Architectural Overview

2.1 System Structure

PC2v9 operates using a *client-server* architecture. Each site in a contest runs a single PC^2 ***server***, and also runs multiple PC^2 ***clients*** which communicate with the site server.[1] The types of clients which might be running in any given contest include the *Admin*, *Application Team* (or just *Team* for short), *Judge*, *Scoreboard,* and *Event Feed* (or *Feeder* for short). An additional client type is the *EWTeam*, a web-based version of the Team client.

Client modules communicate *only* with the PC^2 server at their site; clients *never* communicate directly with each other or with any server other than the one at their site.[2] Communication between clients and their server operates on a "packet transmission" basis. That is, whenever a module (say, an Admin, or a Team) wants to communicate with the server, it does so by constructing a "packet" of information and sending that packet to the destination module using a PC2v9 software layer called the "Transport". (The code which handles transporting packets between modules over the network can be found in package edu.csus.ecs.pc2.core.transport, and is described in further detail in a subsequent section; at this point it is only necessary to understand that when a module wants to send a packet to another module it does so by passing the packet to the "transport layer", which forwards it to the destination module.)

The types of packets which can be sent between modules are defined in class PacketType in package core.packet.[3] An "enum" named Type in that class defines all of the types of packets recognized by the PC2v9 system. For example, one type of packet is a RUN_SUBMISSION packet. When the Team module has a run to submit to the Server to be judged, it constructs a RUN_SUBMISSION packet and passes that packet to the Transport, which forwards it to the Server. The various types of packets and how they are handled in the code is described in further detail in subsequent sections.

2.2 MVC Architecture

Each PC2v9 code module (server, team, judge, scoreboard, etc.) is organized using the *Model/View/Controller (MVC)* architecture. Each module contains a *model*, which holds the data associated with the contest. The various types of model data are defined as classes in package core.model. Each module's model is an instance of class core.model.InternalContest; the model is typically represented in the code by convention as a variable named contest (although this is not 100% consistent and there is no rule which enforces it). Class InternalContest implements an

[1] A *site* in a PC^2 contest is a *logical grouping of clients, together with a single server*, irrespective of physical location. A single "site" might consist of teams and judges in widely-separated physical locations (different cities or countries, for example), all communicating over the Internet with their (single) site server. Alternatively, a contest might be run in "multi-site mode", with different groups of clients each communicating with their own site server (in which case the servers also communicate with each other to maintain the overall state of the contest).

[2] Note however that in some cases the server will forward messages which it receives from one client onward to another client, or to another site's server.

[3] All Java packages comprising PC2v9 start with edu.csus.ecs.pc2.; "core.packet" actually refers to package edu.csus.ecs.pc2.core.packet. For simplicity, the universally common prefix will be omitted from package names in subsequent descriptions.

interface named IInternalContest which defines the methods used to manipulate the model (contest) data. In other words, each PC2v9 module contains its own "internal contest representation" (model) together with a set of predefined methods for manipulating that contest model's data.

Each module also contains a *controller* which manages the manipulation of the contest model. Each controller is an instance of class core.InternalController. The InternalController class implements an interface named IInternalController, which defines the methods available for controlling (manipulating) the model as well as for performing other "control" functions. (For example, IInternalController defines methods such as SubmitRun() for inserting a run into the model; setContestTime() for updating the contest time stored in the model; login() for logging a client into the server, and so forth.)

Finally, each module also has associated with it one or more *views.* A view is (typically) a graphical user interface (GUI) which displays widgets on the screen to allow users to invoke various functions in the controller and hence manipulate the model or perform other actions. Views are classes defined in package edu.csus.ecs.pc2.ui and also within subpackages inside the ui package. For example, when an *Application Team* client module starts it displays the GUI defined by class ui.team.TeamView.

2.3 Packet Handling

When the transport layer in a module receives a packet from the network, it forwards the packet up to the module's controller, which in turn passes it to a method processPacket(). Every module contains an instance of a class named PacketHandler. processPacket() invokes a method handlePacket() in the PacketHandler class. handlePacket() is the place in the code where packets are "dispatched" (sent to the appropriate processing routine).

handlePacket() contains a giant "switch" statement that directs (dispatches) packets to the appropriate place. For example, a RUN_SUBMISSION packet gets dispatched to a method named runSubmission(). The runSubmission() method does some preliminary checking, then tells the "contest" (model) to "accept" the run. The run also gets timestamped (updated with the current time) in runSubmission(), and an acknowledgement-of-receipt-of-run packet is constructed and sent back to the team. The server then sends a packet to the Judges notifying them of the existence of the (new) run.

When a judge module's PacketHandler gets a packet indicating a new run, it turns around and sends a RUN_REQUEST packet to the server. The server then marks the requested run as "checked out" and returns information about the run to the requesting judge. At this point the judge can compile, execute, and validate the run, returning result information to the server, which then forwards the results (again, via a packet) to the team module.

2.4 Listeners

PC2V9 makes extensive use of the *Observer Design Pattern* (also sometimes called the *Observer/Observable* pattern or the *Listener* pattern).

***TODO: add a description here of where in the code *Listeners* are registered and how they work.

3 System Startup

3.1 Module Startup

PC2v9 modules are started by executing a script located in the bin folder inside the PC2v9 Eclipse project (just below the root of the Eclipse project, at the same level as the src folder containing the PC2v9 Java classes). For example, starting a server is done by invoking the pc2server script; starting a Judge is done by invoking the pc2judge script, etc.[4] Although the scripts vary slightly in some details, fundamentally they all do the same thing: invoke the java command to start a Java Virtual Machine (JVM) running a class named edu.csus.ecs.pc2.Starter, passing to Starter any command line arguments specified by the user.

When the Starter class begins running, it constructs a new InternalContest (model), constructs a new InternalController (controller), attaches the model to the controller so that the controller knows what model (contest) it is manipulating, and calls method start() in the controller passing any command line arguments to the start() method.

The controller's start() method first parses the command line arguments using an instance of class core.ParseArguments, which stores all arguments in a hashtable. start() then passes the parsed arguments to the contest (model).

Next, the start() method checks the parsed arguments to determine whether the module being started is a PC^2 Server or a Client module. If the command line contained a "**`--server`**" argument then the module is a Server; in this case start() initializes the server module's "Profile" if necessary (that is, in the case that this server is not going to be obtaining a profile from another remote server). In either case (server or client module), start() also then initializes the module's logging mechanism.

start() then checks for and handles any special command line arguments such as "**`--help`**" (which displays a "usage" message then exits); "**`-F`**", which instructs the module to read additional options from a file; "**`--nogui`**", which instructs the module to run without displaying a Graphical User Interface[5]; and "**`--debug`**", which causes the module to output verbose debugging/tracing information to the console.

Next, start() initializes the module's transport layer by creating a new instance of class core.transport.connection.ConnectionManager (however, the transport layer is not actually started yet; see below).

start() then checks the command line arguments. Unless the arguments include the option "**`--skipini`**", it checks to see if the arguments include "**`--ini`**"; if so, it uses an instance of the class core.Ini to load the **.ini** file whose URL is specified by the argument; if no "**`--ini`**" is

[4] There are two similarly-named versions of each script, one containing Windows (DOS) commands and another containing Unix (Linux/MaxOSX) commands; typing the name of a script at a command prompt in a given OS will automatically invoke the corresponding OS script version.

[5] While most modules run using a GUI, there are some modules (such as the Server) which support a "--nogui" option to allow them to run "headless". See the PC2v9 Administrator's Guide for further details.

present, it uses an instance of the class core.IniFile to load the **.ini** file whose name is the default, **pc2v9.ini**. [6]

After reading the **.ini** file, the start() method starts the ConnectionManager's transport layer running. For server modules, this causes the transport to start listening for packets from other modules (clients as well as remote servers). For client modules, start() also invokes a method connectToMyServer() which establishes a new core.transport.ConnectionHandler for managing connections from the client to its server.

Once the above initialization has completed, the start() method checks whether login/password credential information was specified in the command line arguments and whether the module is to be run in "GUI mode" or not. For the case of GUI mode, start() constructs an instance of class ui.LoginFrame, installs the contest (model) and controller into the frame, and displays the login frame, which prompts the user for an account and password.

Once the user has provided credentials (either having provided them on the command line or through the GUI), the controller's login() method is invoked (either directly by the start() method or indirectly by the action handler attached to the GUI "Login" button). This sends a LOGIN_REQUEST packet containing the credentials. For a client module, this packet is sent to the server whose IP address is specified in the client's **.ini** file; for a server, this packet goes directly back to the packet handler in the same server module. In either case, the server examines the credentials and returns either a LOGIN_SUCCESS or LOGIN_FAILED packet, which the controller uses to determine the next action to be taken (see the section on **Login and UI Frames**, below).

3.2 Login and UI Frames

When a controller's start() method sends a LOGIN_REQUEST packet to a server, it gets back either a LOGIN_SUCCESS or a LOGIN_FAILED packet. Both types of packets are initially handled, as usual, in the module's PacketHandler instance.

A LOGIN_FAILED packet is handled by invoking method InternalContest.loginDenied() in the contest model; this method fires an event of type LOGIN_DENIED which is forwarded to a listener registered at startup with the server by the original requester.

A LOGIN_SUCCESS packet is processed in the controller's PacketHandler, by invoking method loginSuccess(). This method first checks to see if this is the very first login. If so, it checks whether the login came from a Client or from another server (in the case of a multi-site contest). In the case of a login from another server, the method fetches the "contest master password" (defined by the user the first time a server is started) and verifies that it matches the contest master password provided by the packet sender (the remote server). In either case (client or server) the method then initializes the model with the data from the packet.

Once the module (client or server) has been initialized, the loginSuccess() method invokes method otherLoginActivities(), which in turn calls the controller's startMainUI() method passing to it the "client ID" for the module which logged in. startMainUI() first checks if the logged in module

[6] Other than the fact that class core.Ini supports reading from an arbitrary URL while class IniFile always uses a local file, there is no good reason for having two different classes for loading **.ini** files; it is an historical anomaly that should be cleaned up by refactoring.

is a server, and if so it tells the Transport layer to start listening for connections on the port indicated by the remote server.

startMainUI() then checks whether the module is supposed to be using a GUI. If so, and if no GUI has previously been specified (which is the default case), startMainUI() calls static method LoadUIClass.getUIClassName() to find the name of the GUI class which should be displayed for the specified client ID and then calls static method LoadUIClass.loadUIClass() to load the specified UI class. startMainUI() then calls setContestAndController() in the specified UI class, inserting the contest model and the controller into the GUI class (so that the GUI class, which implements the client *view*, will have access to the corresponding MVC model and controller). Finally, if the (no-longer-needed) login frame still exists, startMainUI() disposes it.

GUI class names are defined by LoadUIClass.getUIClassName() in one of two ways. When called, this method first checks for the existence of a "properties file" whose name is given in the System Properties table under the key "uiname". If the file is found, or if there is a folder of the corresponding name and that folder contains a file named "uiname.properties", the file is loaded into a Properties object which is then checked for a GUI property name corresponding to the client. If found, that name is returned as the UI class name for the client.

If no UI properties file is found, or if the file does not contain a key matching the client's name, then getUIClassName() returns a default GUI class name for the specified client. The list of default class names is defined in method getDefaultUIProperties(); each type of client has a default UI whose class name is specified by that method.

4 The Admin Client

4.1 Overview

<TODO: Need here a general description of how the code creates the Admin screen – the Configuration and Run Contest tabs and each of the sub-tabs they contain, along with the MessagePanel common to all sub-tabs.>

4.2 The Problems Tab

The Problems tab contains an instance of class ProblemsPane. This class extends ("is-a") JPanePlugin and contains three GUI components, created by the initialize() method invoked by its constructor: a *message panel* (a JPanel containing a JLabel used to display messages); a *center panel* (a JPanel containing a Multi-Column List Box (MCLB) displaying a grid showing each defined contest problem), and a *button panel* (a JPanel containing buttons Add, Copy, Edit, Report, and Set Judge's Data Path).

The initialize() method also creates instances of separate GUI classes EditProblemFrame and EditProblemFrameNew (the latter is only used in debugging mode). EditProblemFrame is a JFrame containing one thing: an instance of class EditProblemPane. EditProblemPane in turn contains the GUI components needed for editing a problem (including creating a new Problem): a *message pane* at the top; a *button pane* at the bottom containing buttons Add, Update, Report, Load, Export, and Cancel; and a *main tabbed pane* containing four tabs: General, Judging Type, Validator, and Data Files.

Note: the buttons on the EditProblemPane button pane are not always visible. If you open the code in a GUI Builder such as "Window Builder Editor" in Eclipse it will show all six buttons; however, which buttons are visible in the running code at any time depends on the state of the system at that time. For example, the "Update" button is not visible when adding a new problem; similarly, the "Add" button is not visible when updating an existing problem.

Pressing the Add button on the ProblemsPane (not the EditProblemPane) invokes a method setProblem(Problem) in the EditProblemFrame, passing null as the Problem value. When setProblem() receives **null** as the problem value it sets the title of the EditProblemFrame to "Add New Problem", then invokes method setProblem(Problem) in the EditProblemPane. This method clears the data files on the Data Files tab, then invokes method populateGUI(Problem).

Pressing the Edit button on the ProblemsPane (after selecting a Problem in the MCLB problem grid) invokes method setProblemCopy(Problem,ProblemDataFiles) in the EditProblemFrame, passing to it the selected problem and the corresponding problem data files. setProblemCopy() in turn invokes setProblem(Problem,ProblemDataFiles) in EditProblemPane (analogous to the way the Add button invokes EditProblemPane.setProblem(Problem)). setProblem(Problem,ProblemDataFiles) then clears the data files on the Data Files tab, load the specified ProblemDataFiles into the Data Files tab, initializes the Data Files tab GUI components by calling method MultipleDataSetPane.populateUI(), and finally calls method populateGUI() in the EditProblemPane. Thus, both the Add button and the Edit button end up calling populateGUI(), passing either **null** (when adding a new problem) or a specific Problem (when editing an existing problem).

populateGUI() is the method responsible for loading problem into the EditProblemPane GUI. When populateGUI() receives **null** as the Problem value, it invokes method clearForm() to load the EditProblemPane GUI with the initial defaults used for defining a new problem. If populateGUI() receives a non-null Problem, it instead invokes method setForm() to populate the GUI from the values in the received problem. When the GUI has been populated from a received problem, populateGUI() then invokes method getProblemFromFields() to insure that the resulting populated values are sane (getProblemFromFields() throws Exception InvalidFieldValue if any of the populated values are invalid). Finally, populateGUI() enables the various EditProblemPane components as appropriate, loads the appropriate data file values into the Data Files tab, and makes the "General" tab the currently selected tab.

5 The EWTeam Client

5.1 Overview

"EWTeam" is a PC2v9 module that allows contest teams to connect to a PC^2 server using a web browser. It displays a web page that allows teams to submit runs and clarification requests and to receive responses from the judges, just like the "Application Team" client. In addition it allows the team to examine the current scoreboard (a function not provided by the Application Team client). The principle advantage of the EWTeam client is that it does not require installing anything on the team machine; all that is needed to access the PC^2 contest is a web browser.

The EWTeam module was originally developed as a separate project by students at Eastern Washington University (hence the name). It has subsequently been merged into the PC2v9 project, housed under the pc2v9/projects folder in Eclipse and stored in Git just as with the rest of the PC2v9 system.

Architecturally, the EWTeam consists of a collection of modules written in different languages. The front-end web pages are handled by a set of PHP modules which must be installed on the web server. The PHP code invokes routines in a module called the *Java Bridge* (which is started by a class named PC2JavaMiniServer and must be running before the EWTeam web pages can be used). The Java Bridge allows PHP code to make calls into Java code; specifically, the PHP code calls routines in class ServerInterface. The ServerInterface methods in turn make calls to an embedded copy of the PC^2 API contained in a jar file. The API methods in turn communicate with an instance of the PC^2 server for the site.

5.2 Project/Code Organization

TODO: add text here describing the layout and purpose of each of the files/folders in the EWTeam project.

5.3 Run Submission

The EWTeam PHP code which handles the "Submit" button on the web page invokes a method submitProblem() (in class ServerInterface in the default package in the EWTeam project /src folder) via the Java Bridge. The submitProblem() method in turn invokes the PC^2 API method api.ServerConnection.submitRun(), which invokes method core.InternalController.submitRun(). All PC^2 applications that submit a run use core.InternalController.submitRun() to submit the run to the same packet-handling entry point in the server -- so runs submitted via the web interface are no different from runs submitted via the Team Application client as far as the Server is concerned.

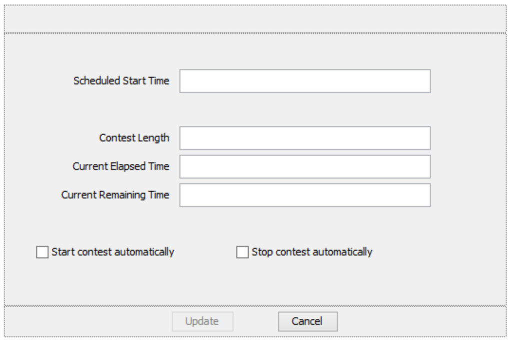
6 Contest Time

6.1 General Description

TODO: explain how the code handles the concepts surrounding setting and updating of ContestTime objects…

6.2 AutoStarting

A PC2v9 contest can be configured to "automatically start" when a specified date/time has arrived. Automatic starting is handled via the combined action of several different classes. To begin with, classes ui.AdministratorView and ui.ServerView both add instances of class ui.ContestTimesPane to the "Times" tab in their respective JTable displays. ui.ContestTimesPane displays a grid with one row for each site in the contest, along with a set of buttons for managing "time" at whichever site row in the grid is currently selected. One of the buttons is "Edit", which supports editing of the time information for the selected site by popping up a new instance of class ui.EditContestTimeFrame containing an instance of class ui.EditContestTimePane, which looks like the following:



If the user enters a future date and time in the "Scheduled Start Time" field, checks the "Start Contest Automatically" box, and clicks "Update", the following things related to automatic contest starting happen in method handleUpate() (that is, the Action handler for the Update button) in class EditContestTimePane:

1. The scheduled start time in the GUI textbox is checked to insure it matches the valid format, which is yyyy-mm-dd HH:mm.
2. The scheduled start time (if it is valid) is checked to insure it is in the future.
3. The "Start contest automatically" checkbox is examined to see if it is selected (checked).

If all the above conditions are satisfied, the specified "Scheduled Start Time" is stored into ContestInformation object, along with the settings of the Start Contest Automatically and Stop Contest Automatically checkboxes. handleUpdate() then invokes method updateContestInformation() in the controller. This method in turn sends a new UPDATE_SETTING packet to the server containing a new ContestInformation object.

When the server's PacketHandler instance receives an UPDATE_SETTING packet containing a ContestInformation object, it invokes method updateContestInformation() in the local contest (model), then checks to see if this packet is being handled on a Client or a Server. If this is a Server, the packet handler checks to see whether there is a Scheduled Start Time in the received ContestInformation object. If so, and if the Scheduled Start Time is in the future and also the ContestInformation object indicates that the "Start Contest Automatically" box had been checked, then the packet handler invokes a controller method scheduleFutureStartContestTask().

The scheduleFutureStartContestTask() method verifies that the specified "start time" is in the future. If so, it creates a new java.util.concurrent.ScheduledExecutorService object, which it designates as the scheduler. Next it constructs a new java.lang.Runnable whose run() method calls either the controller's startContest() method or its startAllContestTimes() method.

scheduleFutureStartContestTask() then inserts this Runnable into the scheduler with an execution time based on the "Scheduled Start Time". The effect of this is that when the scheduled future time arrives the Java scheduler object will execute the Runnable, which will invoke the startContest() method (or the startAllContestTimes() method) in the controller.

Regardless of which of these two methods is called, the effect is to send a packet (either a START_CONTEST_CLOCK packet or a START_ALL_CLOCKS packet) to the local server. When the server's packet handler subsequently receives this packet, it calls method core.PacketHandler.startContest(). This method in turn calls startContest() in the model (InternalContest), which marks the clock as started and then fires an event of type CLOCK_STARTED.

***TODO: currently in the code the Runnable which auto-starts the contest calls controller.startContest(), which eventually causes a START_CONTEST_CLOCK event to be fired when in reality what needs to be fired is CLOCK_AUTO_STARTED. Need to fix this in the code, then update the following to match...

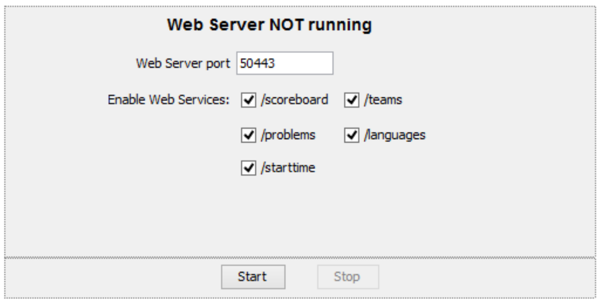
The CLOCK_AUTO_STARTED event is passed to instances of IContestTimeListener which are registered to listen for ContestTimeEvents at system startup; each module which wants to know when some ContestTimeEvent has occurred registers itself to listen for such events (or more correctly, registers an instance of its local implementation of IContestTimeListener as a listener). For example, the AdministratorView class registers an IContestTimeListener when it starts, so the occurrence of the Clock_Auto_Started event invokes the contestAutoStarted() method in the AdministratorView; similar registrations and corresponding invocations occur with each other module that wants to be notified when the contest has been (automatically) started.

In most cases the contestAutoStarted() method in IContestTimeListener implementations simply delegate the handling of the Clock_Auto_Started event to the contestStarted() method in the same listener – the effect being that auto-starting a contest is the same as manually starting it. There are however in some IContestListeners additional steps that are taken by the contestAutoStarted() method. For example, the AdministratorView contestAutoStarted() method also pops up a message box on the Admin GUI notifying the user that the contest has automatically started.

6.3 AutoStarting via the Web Interface

PC2v9 contains a web server which can be used to access various data via a browser. The data includes information on the current Teams, Languages, Problems, and Scoreboard. In addition, the web interface can be used to control the auto-start facility.

The PC2v9 web server does not automatically run; it must be manually started. This is done from an instance of class ui.WebServerPane (shown below), which appears on the Event Feed client screen:


Web Server NOT running

Web Server port 50443

Enable Web Services: ✓ /scoreboard ✓ /teams ✓ /problems ✓ /languages ✓ /starttime

The actionPerformed() method for the **START** button on this pane invokes a method startWebServer() in the WebServerPane class. This method invokes a factory method getWebServer(), which constructs an instance of the class services.eventfeed.WebServer and returns it. The actionPerformed() method then invokes startWebServer() in the new WebServer object, passing it the current controller, model, and a Property object that includes the states of all the "Enable Web Services" checkboxes on the GUI.

The startWebServer() method initializes a java.security.KeyStore (security keystore) object for the web server, then constructs a *Jetty*[7] server instance. It configures Jetty with a ServletContextHandler into which it inserts a *Jersey*[8] ServletContainer holding a Jersey ResourceConfig object containing instances of each of the enabled PC2v9 web services. For example, if the "/teams" checkbox was checked, the Jersey ResourceConfig will hold an instance of class services.web.TeamService; if "/starttime" was checked it will (also) hold an instance of class services.web.StarttimeService; etc. The objects contained in the ResourceConfig installed into the Jetty ServletContainer define what web services (REST endpoints) the PC2v9 web server recognizes.

Each of the implemented web service REST endpoints are defined in package services.web and (optionally) configured into the web server in method services.eventfeed.WebServer.getResourceConfig(). Once the Jetty web server is configured and started, it listens on the specified port for HTTPS connections, performs security checks on connection requests, and forwards those requests which pass the security checks to the appropriate GET or PUT method in the corresponding registered service, one of which is the StarttimeService.

The StarttimeService GET method simply obtains the currently-defined start time from the contest model, formats it as a JSON string of the form **{"starttime":<time>}**, and returns that string as the HTTP response, where <time> will contain a number representing the Unix Epoch start time (the number of milliseconds from January 1st, 1970) or will contain the string "undefined" if no scheduled start time is currently set.

The StarttimeService PUT method accepts a JSON string in the above format. It first checks it for a variety of conditions (verifying the format is valid, checking to see that the time, if

[7] *Jetty* (http://www.eclipse.org/jetty/) is an embeddable, configurable web server which provides the web server implementation in PC2v9

[8] *Jersey* is an implementation of the JAX-RS (Java API for RESTful Services) specification (see https://jersey.java.net/ and https://jax-rs-spec.java.net/ for further information

specified, is in the future and meets a variety of other conditions (such as not being too close to an already-scheduled start time) as specified by the CLICS Starttime Interface specification at https://clics.ecs.baylor.edu/index.php/Draft_2014_REST_interface_for_source_code_fetching.

If the start time is valid, the StarttimeService PUT service calls controller method updateContestInformation() passing to it an updated ContestInformation object containing the new scheduled start time (which could be a time or could be the string "undefined", which is used to tell the system to remove any scheduled start time). At this point the code path becomes the same as that described in "AutoStarting", above: the controller sends an UPDATE_SETTING packet to the server, and so forth. In this way, invoking the StarttimeService PUT web service acts the same as a user invoking "Start Contest Automatically" from the EditContestTimePane, above.

7 The Transport Layer

7.1 Overview

TODO: add text here describing the overall structure of the Transport Layer…

8 New Section

8.1 New Subsection

Some text….

Appendix A – Some Additional Stuff

Some text to be determined….