



# ACM INTERNATIONAL COLLEGIATE PROGRAMMING CONTEST



California State University, Sacramento's

**PC<sup>2</sup>**  
Version 9

# Technical Specifications

<Last Update: 2016-09-21>

## Table of Contents

<b>1</b>	<b>Introduction .....</b>	<b>2</b>
<b>2</b>	<b>Architectural Overview .....</b>	<b>3</b>
2.1	System Structure .....	3
2.2	MVC Architecture .....	3
2.3	Module Startup .....	4
2.4	Packet Handling .....	5
<b>3</b>	<b>The EWTeam Client .....</b>	<b>6</b>
3.1	Overview .....	6
<b>4</b>	<b>Contest Time .....</b>	<b>7</b>
4.1	General Description .....	7
4.2	AutoStarting .....	7
<b>5</b>	<b>The Transport Layer .....</b>	<b>9</b>
5.1	New Subsection .....	9
<b>6</b>	<b>New Section .....</b>	<b>10</b>
6.1	New Subsection .....	10
	<b>Appendix A – Some Stuff .....</b>	<b>11</b>

# 1 Introduction

PC<sup>2</sup> 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<sup>2</sup> 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
<b>src</b>	Java source code for the various system components
<b>test, testdata, and testout</b>	a variety of Java JUnits for regression testing, along with the data used by those tests and the output produces by running the tests
<b>bin</b>	scripts used for module startup and various administrative operations
<b>data</b>	XSL files for generating formatted scoreboards
<b>dist</b>	current distribution versions of various modules
<b>doc</b>	PDF and MS Word (source) versions of various system documents
<b>images</b>	image files used by various modules
<b>projects</b>	system components which were organized for historical reasons as separate projects (but which are now integrated into the PC2v9 system),
<b>samps</b>	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.
<b>vendor</b>	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<sup>2</sup> system. For further information on PC<sup>2</sup>, including how to configure the parameters which control and enable various judging operations, refer to the *PC<sup>2</sup> Contest Administrator’s Installation and Configuration Guide*, which can be found at <http://pc2.ecs.csus.edu/>.

## 2 Architectural Overview

### 2.1 System Structure

PC2v9 operates using a *client-server* architecture. Each site in a contest runs a single PC<sup>2</sup> *server*, and also runs multiple PC<sup>2</sup> *clients* which communicate with the site server.<sup>1</sup> 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<sup>2</sup> server at their site; clients *never* communicate directly with each other or with any server other than the one at their site.<sup>2</sup> 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`.<sup>3</sup> 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 interface named `InternalContest` which defines the methods used to manipulate the model (contest)

---

<sup>1</sup> A *site* in a PC<sup>2</sup> 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).

<sup>2</sup> 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.

<sup>3</sup> 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.

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 a *view*. This 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 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.<sup>4</sup> 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`, 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 parses the command arguments to determine what it is supposed to do, then initializes the system logging mechanism, the module’s transport layer communications channel, and the model as appropriate.

Next, the `start()` method checks the command arguments to determine whether the module being started is a PC<sup>2</sup> Server or a Client module. If the module is a Server then it checks to see whether there are other servers (sites) in the contest and if so it attempts to connect to a specified remote server. In any case the Server then starts listening on a specified port for information (packets) from other modules.

If the module being started is instead a Client of some sort then the controller’s `start()` method constructs an `ILoginUI` frame, installs the contest (model) and controller into the frame, and displays the login frame which asks the user for an account and password. Once the user provides valid credentials the controller discards the login UI frame and starts the main UI for the type of

---

<sup>4</sup>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.

client whose credentials were provided. The controller then waits for either UI input from the user or for receipt of a packet from another module via this module's transport layer.

## 2.4 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.

## 3 The EWTeam Client

### 3.1 Overview

“EWTeam” is a PC2v9 module that allows contest teams to connect to a PC<sup>2</sup> 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 PC2 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. (Actually, what is stored in the projects folder is a compressed version of the separate EWTeam project; unzipping either the .zip or .tar.gz file

TODO: add text here describing the architectural differences of the EWTeam – the PHP web server code, the transition to the PC2API, and the similarities (and differences) of how a “RUN\_SUBMISSION” packet gets to the server from the EWTeam as opposed to the Application Team client.

TODO: add more description of the EWTeam process, such as “The PHP code which handles the “Submit” button on the web page invokes a method “runSubmitted()” in the “PC2API” package, which in turn invokes 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.” Or something similar... to be completed....

Note from Doug: Small correction, EWTeam invokes: `ServerConnection.submitRun` in package `edu.csus.ecs.pc2.api` which then invokes `controller.submitRun` just like the pc2 team GUI does, where controller is `IInternalController` controller

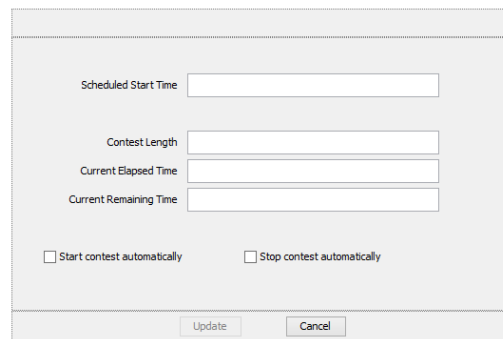
## 4 Contest Time

### 4.1 General Description

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

### 4.2 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 `handleUpdate()` (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 the current contest (model)’s `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 contest (model).



This method (in class **InternalContest**) 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 **updateContestInformation()** invokes a 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 does three things:

- a) Call method **startContest()**
- b) Construct a new **core.model.ContestTimeEvent** of type **Action.Clock\_Auto\_Started**
- c) Fire the new **ContestTimeEvent**

**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 and fire the **Clock\_Auto\_Started ContestTimeEvent**.

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.

## **5 The Transport Layer**

### **5.1 New Subsection**

Some text....

## **6 New Section**

### **6.1 New Subsection**

Some text....

## **Appendix A – Some Stuff**

Some text....