

# The Design of the CEL System.

Elizabeth Sklar  
DEMO Lab  
Department of Computer Science  
Brandeis University  
Waltham MA 02454-9110  
*sklar@cs.brandeis.edu*

### **Abstract**

This report describes the design of the CEL (Community of Evolving Learners) system. CEL is a learning community on the Internet where students engage in simple multi-user educational activities. This document is intended to serve as software documentation for anyone intending to add activities to the system as well as maintain, replicate or expand on the infrastructure. It is assumed that the reader is familiar with the UNIX environment and has a working knowledge of ANSI C and Java version 1.0.2.

# Chapter 1

## Introduction.

The Community of Evolving Learners (CEL) is an Internet community where students engage in simple multi-user educational activities. To date, these are two-player educational games. With CEL, our aim is to break the traditional barriers of classroom walls and link students with similar abilities but diverse ages, genders and locations. The system was designed to be accessible, flexible and extensible, to support a variety of experiments in both human and machine learning.

The purpose of this document is to serve as software documentation for anyone intending to add activities to the system as well as maintain, replicate or expand on its infrastructure. It is assumed that the reader is familiar with the UNIX environment and has a working knowledge of ANSI C and Java version 1.0.2.

The CEL system is based on a client-server architecture. It is written in Java version 1.0.2 and ANSI C. It uses sockets for communication between the server and all clients. In this way, communication is assumed to occur over the Internet using standard protocols.

The network model for CEL uses one centralized server and any (reasonable) number of clients. The clients communicate with the server and with each other, indirectly through the server. All communication uses a special language, called the CEL Message Language (CELML).

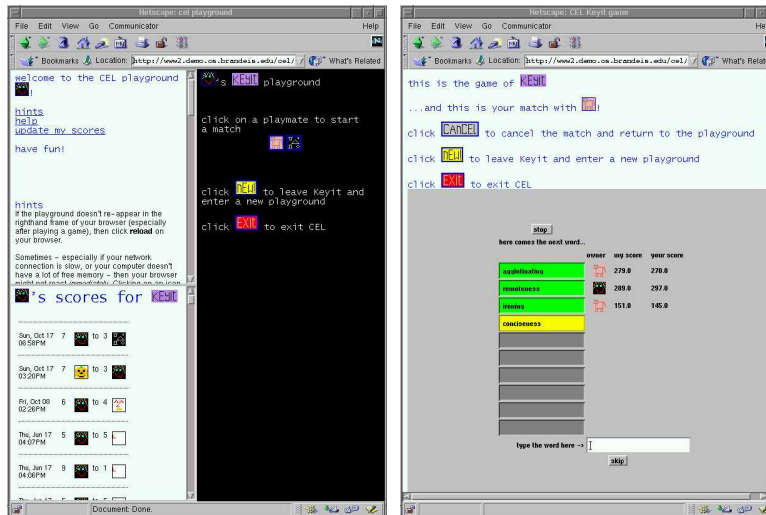
This report details the design of the CEL system. The document is organized as follows. Section 1.1 contains a brief overview of CEL, giving a brief tour of the system from a user point-of-view and introducing some terminology.

A note about terminology and formatting in this document: words that are Java keywords are highlighted in **this font**; words that are CEL keywords (e.g., CEL classes, variables and commands) are highlighted in **this font**.

## 1.1 System Overview.

CEL is located on a free web site and is open to anyone with Internet access and a web browser<sup>1</sup>. Students log into CEL with a personal user name and password. Once inside, participants are represented by two-dimensional graphical icons called *IDsigns* which identify them to other users. Their user names (and passwords) are never shown to others. A tool called the *IDsigner* is employed to create and edit IDsigns.

After logging in, users select an activity to engage in and then they are placed in a virtual *playground* (see figure 1a). Here, a graphical matrix is displayed containing the IDsigns of each user's *playmates* — others who are currently logged into CEL and are involved in the same activity. By clicking on a playmate's IDsign, a student can invite a playmate to engage in a match. The match begins when the browser displays a *game page*. The players participate asynchronously. When the match is over, each player is returned to her playground and is then free to engage in another match with another (or the same) playmate.



(a) A CEL playground.

(b) Keyit.

Figure 1: CEL screens.

Currently, the activities in CEL are simple, two-player educational games designed to reinforce skills introduced in primary classrooms. Thus far, several word and math games have been created, exercising basic skills such as keyboarding, spelling and arithmetic.

The word game called *Keyit* is pictured in figure 1b, and a close-up is shown

<sup>1</sup>Netscape is currently the only browser fully tested.

in figure 2. In this competitive activity, participants are each given 10 words to type as fast as they can, maintaining 100% accuracy. For each player, a timer begins when she enters the first letter of a word and time is marked when she presses the *Enter* key to terminate the word. Time is measured using the system clock on the user’s computer, and score is reported in hundredths of a second.

During the course of a match, feedback is provided to both players by filling in the time as words are completed, in the column designated for each player. The “owner” field is updated with the IDsign of whichever player typed each word faster.




	owner	my score	your score
extend		105.0	265.0
middle		200.0	86.0
par		25.0	24.0

Figure 2: A Close-up of Keyit.

The players need not be synchronized, nor even participate simultaneously. For example, a network link may be slow or one user may be interrupted. In these instances, the system provides to each user whatever moves are available from the opponent. Sometimes this means that a user does not get complete participation from her opponent — she may finish a game before her opponent has finished or even started.

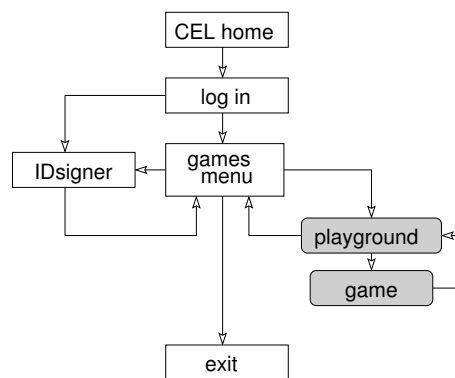


Figure 3: Site Map.

A map of the CEL site is shown in figure 3, illustrating the relationship between menu, playground and game pages. The unshaded boxes represent the static portion of the site. The shaded boxes show the virtual position of a single

playground and game page. For each activity in CEL, there are pairs of playground and game pages. These are created dynamically, as users enter and exit playgrounds and initiate matches.

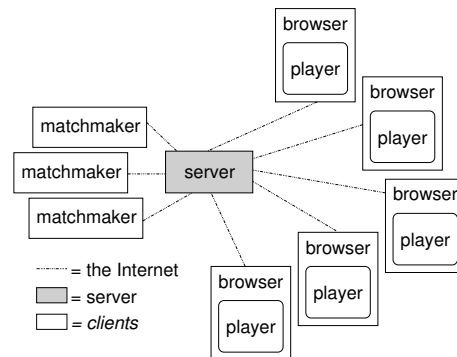


Figure 4: System Architecture.

The CEL system employs a client-server architecture (see figure 4). One central *server* maintains a dynamic database indicating who is logged into the system and which activities they are engaged in. This server also acts as a message passer, sending and receiving commands to and from clients. For each activity in CEL, a *matchmaker* keeps track of the users that are currently active and their game state — either “lonely” or playing a game.

The *player* is designed to meet two fundamental needs: (1) to be practicable to anyone with Internet access and a web browser<sup>2</sup>, and (2) to be usable by students with limited network bandwidth and low computer memory. As such, we use small footprint Java applets for games and implement the playgrounds using C programs that generate HTML and refresh periodically in order to update playgroup content.

The system is modular and extensible. A new activity is added to CEL by extending Java classes to implement a matchmaker and a game applet. Our longterm plan is to place these classes and documentation on our web site in order to allow others to contribute to the community by creating their own activities. Although all matchmakers currently reside on our server, matchmakers could be instantiated on any networked computer, so contributors could choose to host their activities and collect data locally if desired.

---

<sup>2</sup>Netscape is currently the only browser fully tested.

## Chapter 2

# Software documentation.

### 2.1 Directory Structure.

The CEL server is a dual processor Linux computer. Currently, we are running Apache/1.3.12 (Unix) (Red Hat/Linux).

The home directory for CEL is: `/home/httpd/html/cel`. Throughout this document, we will refer to the home directory for CEL as `$CEL_HOME`. Note that it is convenient to define this as an environment variable.

Under the `$CEL_HOME` directory, there are five categories of subdirectories, as described below.

1. source and executable CGI and HTML files for CEL playground:

```
bin/  
include/  
lib/  
src/
```

2. source and class files for CEL framework:

```
bond/  
client/  
idsigner/  
messenger/  
playground/  
matchmaker/  
server/  
util/
```

3. source and class files for CEL activities:

```
Automath/  
Keyit/  
Loois/  
Monkey/  
Puzzlematch/  
Spellbee/
```

4. run-time data and log files:

```
data/  
idsigns/  
logs/  
scripts/
```

5. documentation files:

```
doc/  
javadoc/
```

We are using the revision control system, RCS. Thus, under each directory listed above that contains source code, there is an `RCS/` directory containing the version controlled source files. Explicitly, this means the `src/` subdirectory in category 1, and all subdirectories in categories 2 and 3. Released versions of CGI program files (named `*.cgi`) are placed in the `bin/` directory. Released versions of library files (named `*.a`) are placed in the `lib/` directory. Released versions of header files (named `*.h`) are placed in the `include/` directory.

### 2.1.1 Source and executable CGI and HTML files for CEL playground

There are four subdirectories in this category:

```
bin/  
include/  
lib/  
src/
```

The `src/` directory contains all the source code files. The remaining directories are filled during an installation procedure which is invoked by the `makefile` in the `src/` directory.

The main entries of the `src/` directory are the C source files (i.e., `*.c` and `*.h` files) that are used to build several CGI programs. These are:

- `cel_login.cgi`, which implements the CEL login procedure
- `cel_playground.cgi`, which implements the CEL playground



- `cel_help.cgi`, which displays the CEL help pages
- `cel_quiz.cgi`, which implements an on-line survey

The executable CGI files are placed in the `bin/` directory once they are built in the `src/` directory, using an install script.

Most of these interface with HTML files (listed below), where the HTML files contain templates for what is displayed on the screen. Rather than having the CGI's generate all the HTML code, the CGI's read the HTML template files and make substitutions for variables that are placeholders in the templates. This way, we can change the look-and-feel of the system by changing the templates and only need the CGI's to fill in the current user and game information at run-time.

For example:

```
-----
<p>
click
<a href=''#thisURL#?26&#id#&#tk#&#game#&19&#matchcancel#''
target=''_top''><img src
='http://satchmo.cs.columbia.edu/cel/bin/new.gif''></a>
to leave #game# and enter a new playground
-----
```

The above is taken from the `game.html` template file. The variables that are substituted for at runtime are surrounded by `#` signs:

- `#thisURL#` is the URL of the CEL game page
- `#id#` is the user ID number of the current user
- `#tk#` is the user's session ID (or ticket) number
- `#game#` is the name of the game, e.g., "keyit"
- `#matchcancel#...` i can't remember what that is...

Below are some explanations about some of the CGI files...

### **cel\_login.cgi**

The main source file for the CEL login program is `cel_login.c`. The login procedure is as follows:

1. Initialize log file (`/home/httpd/html/cel/logs/login.log`) so that all error messages (anything output to `stderr`) goes to this file. Note that anything output to `stdout` will go to the `httpd` error log file (`/var/log/httpd/error_log`).

2. Grab the contents of the `QUERY_STRING` environment variable. This contains the value of the full URL which is this login program, e.g.,:  
`http://www2.demon.cs.brandeis.edu/cel/bin/cel_login.cgi`
3. Save the program name (the value of `argv[0]`), which will be used for logging.
4. Initialize program constants.
5. Log into CEL user database.
6. Perform login, invoking standard login facility
7. Start CEL

#### **cel\_playground.cgi**

this is the main source file for when a user is in “playground” mode

#### **cel\_help.cgi**

this is the source file for when a user clicks on the “help” button

#### **cel\_quiz.cgi**

this is the little program that runs periodically to get feedback from the user. it asks two questions: (1) whether the user felt questions were easy or hard (to get at whether the system is challenging the user) and (2) whether the user was having fun or bored (to get at whether the system is motivating the user).

There are also a number of HTML files that are used in conjunction with the CGI files...

*notes below added 17-feb-2007/sklar*

some of these pages are templates that are filled in by a CGI program (probably described above)

- **about.html**  
when the user clicks on “read about the CEL project”
- **consent.html**  
when the user needs to complete the “informed consent” portion of registering with the system
- **error.html**  
when there is a system error during execution
- **exit.html**  
when the user exits the system
- **game.html**  
i think this is the template for the page that is displayed when a user wants to play a game...
- **help.html**
- **help0.html**  
something about asking for help, but i can’t remember exactly  
i think “help” is the template and “help0” is when a template isn’t needed
- **idsigner.html**  
template for the page when the user invokes the IDsigner during the registration process or later when they want to change their IDsign
- **index.html**  
the default home page
- **login.html**  
the login page (template)
- **login\_cancel.html**  
page displayed when the user cancels the login process
- **login\_error.html**  
page displayed when an error occurs during the login process. i think this is a template...

- **login\_retry.html**  
page displayed when the user makes an error occurs during the login process and is asked to retry, like re-enter your username or password. i think this is a template...
- **menu\*.html**  
there are different versions of this, like menu0.html and menu1.html (from 0..7, and A). each one shows a different combination of CEL games. these are substituted for the “live” one, depending on which combination of games we want to be active.
- **menuframe.html**  
the page defining the frame where the games menu shows up...
- **offline.html**  
the page displayed when we take CEL down for maintenance
- **playframe.html**  
the page defining the frame when the user is in the playground. probably a template.
- **playground.html**  
the template page for when the user is in playground mode.
- **pregame.html**  
the template page for when the user is in “pre-game” mode, i.e., they have invited a playmate to play a game and are waiting for the system to respond.
- **quiz.html**  
the template page for the quiz described above (are we having fun? are we challenged?)...
- **signup3.html**  
the registration page for when the user registers with the system for the first time
- **thanks.html**  
the “thanks for taking the quiz” template
- **thanks1.html**  
the “thanks for participating in cel” template
- **view\_consent.html**  
the default page to display the consent form in case someone wants to know what is in it before registering

### 2.1.2 Source and class files for CEL framework

most, if not all, of these components are detailed later in this document. this section just tells where the files are...

- **bond/** directory  
java source files for “secret agents”
- **client/** directory  
java source files for generic Client class. this is extended by the games, the monitor, messenger, etc.
- **dbmanager/** directory  
this is where the database manager source code and startup script go.  
*note:*  
on agents, in `/proj/agents/dev/cel/dbmanager`, only the startup script is in that directory. there is C source code in `/proj/agents/dev/old/dbmanager`. i’m not sure what the status of that is...  
the database manager was created to buffer database updates so that the system wouldn’t hang while database updates were made in case database access was slow. i don’t remember if this was ever finished...
- **server/** directory  
java source files for the Server application.
- **idsigner/** directory  
java source files for the IDsigner applet
- **messenger/** directory  
java source files for the messenger application

#### *NOTE:*

there is a “monitor” program which is described somewhere below. the source for this used to be in the `monitor/` directory, which for some reason doesn’t appear on `/proj/agents/dev/cel`. however, there is a version here:  
`/proj/agents/dev/cel/archive/29may1999/monitor`  
which is probably worth looking at.

### 2.1.3 Source and class files for CEL activities

some games:

- **Automath/**  
directory containing java source files for the AutoMath game, described below
- **Keyit/**  
directory containing java source files for the Keyit game, described below

- **Loois/**  
directory containing java source files for the Loois game, described below
- **MathTree/**  
directory containing java source files for the MathTree game, described below
- **Mathtree/**  
directory containing the C source files for the “kfetch” program that generates math problems and is used by MathTree (above)
- **Monkey/**  
directory containing java source files for the Monkey game, described below
- **Pickey/**  
directory containing java source files for the Pickey game, described below
- **Spellbee/**  
directory containing java source files for the Spellbee game, described below

#### 2.1.4 Run-time data and log files

- **data/**  
this is where the CEL data files are stored—the data files are those that help define games, like the dictionary of words.
- **idsigns/**  
this is where the GIF files for user’s actual IDsigns are stored
- **logs/**  
this is where the CEL log files are stored

#### 2.1.5 Other files

- **backup**  
script to backup the CEL source files
- **buildit**  
script to build the CEL java classes and the C source files. note that some lines are commented out with # in order to selectively perform functions like building C source and checking files out of RCS.
- **backupdata**  
script to create a backup of the data files

- **backuplogs**  
script to create a backup of the log files
- **cel\_offline.sh**  
script to take the system off-line (bring down the server, etc)
- **cel\_online.sh**  
script to bring the system on-line (bring up the server, etc)
- **installit**  
script to create a TGZ file of scripts and class files and then SCP that file to another machine (called “calliope” here, an old machine from Brandeis...)

## 2.2 Home

The CEL home page is: <http://www.demo.cs.brandeis.edu/cel>.

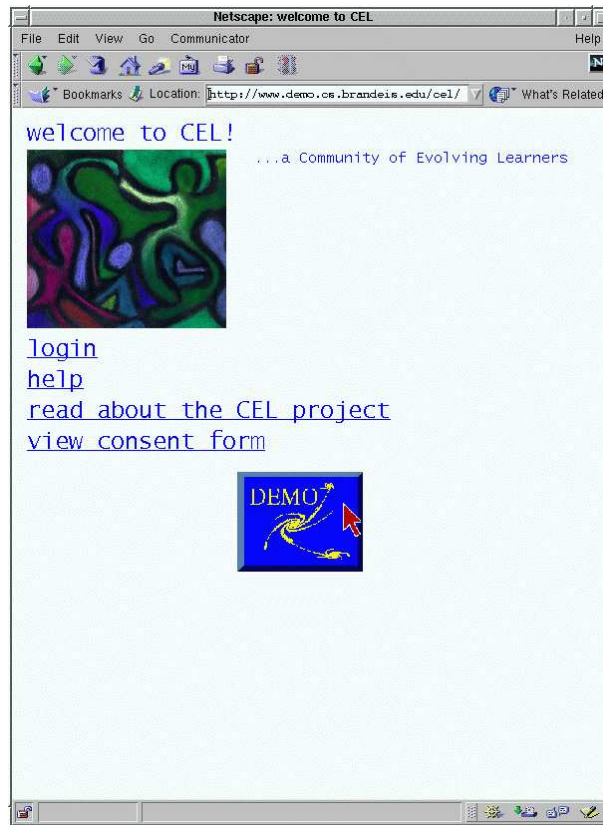


Figure X: CEL home page.

There are four links on this page:

1. login
2. help
3. read about the CEL project
4. view consent form



## 2.3 Login

The CEL login facility is implemented via CGI-bin and is written in C. There are four basic steps involved:

1. Read and validate username and password
2. Verify that consent has been given
3. Verify that user has an IDsign
4. Optionally request user demographic information

The usernames and passwords are stored in a SQL database. Currently (AT BRANDEIS), Postgres WAS used.

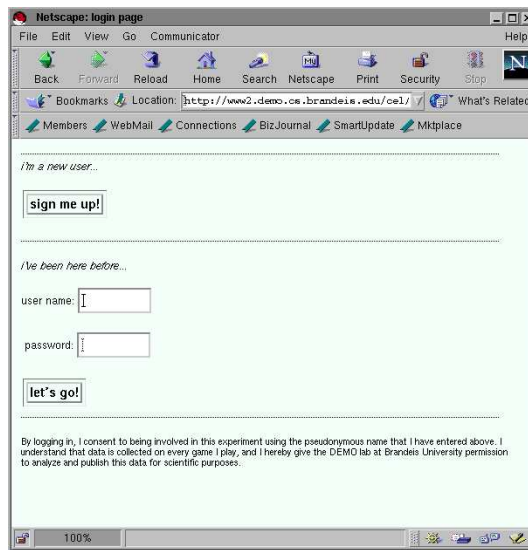


Figure X: Login procedure.

## 2.4 IDsigner

Currently, the primary outlet for creativity in CEL is through users' IDsigns. Each IDsign is  $20 \times 20$  pixels in size. First-time users must create an IDsign during the login process, before they can enter a playground. Participants may also edit their IDsigns later.



Figure 6: IDsigner.

In order to create and modify IDsigns, users access a tool called the *IDsigner* (see figure 6)<sup>1</sup>. Users are given a palette of 13 colors to choose from<sup>2</sup> and a straightforward point-and-click interface with which they can set the color of each of the 400 pixels. IDsigns are saved on our server, so when users return to CEL, the most recent version of their IDsign is loaded automatically.

---

<sup>1</sup>The IDsigner is similar in operation to the KidPix<sup>®</sup> stamp editor.

<sup>2</sup>the standard set of web-safe colors defined in the Java class `java.awt.Color`

## 2.5 Games menu.

Once logged into CEL, the user must pick a game to play. Then she will be placed in the corresponding playground.



Figure X CEL games menu.

## 2.6 Playground.



Figure X CEL playground.

## 2.7 Game.

see description of individual games below, like Keyit, Automath, etc.

## 2.8 Player States.

For all players that are active in CEL, the Server maintains an entry in the `players` vector which includes a `playstate`. The `playstate` can have one of five values: `ENTERING`, `LONELY`, `PREGAME`, `GAME` or `EXITING`. All changes to the `playstate` are initiated by the player, when a challenge is made, a playground update is requested, or the user exits a playground or CEL.

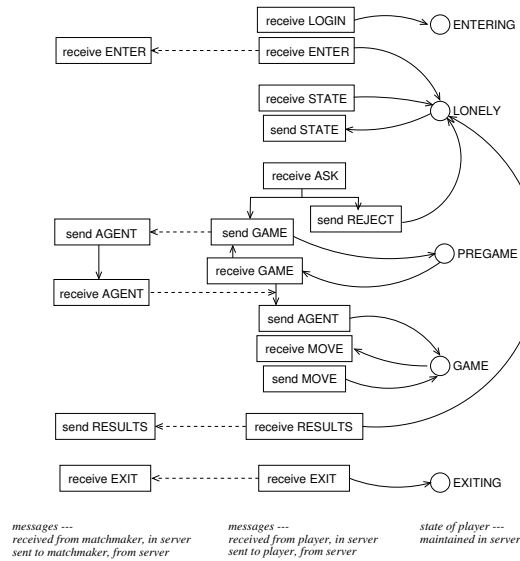


Figure X State diagram for playstate.

Figure X summarizes the state transitions for players inside the server. It shows the normal flow between the five states and the expected sequence of operations. It is possible that the Server and a player can become out of synch, because the player is enabled inside a browser and a user could easily click on a navigation button to change the page and visit CEL pages out of sequence. For example, in the middle of a game, the user could use the browser to visit a playground that is several pages back in the history list. In this case, the Server thinks that the user is in a `GAME` state, but upon receiving a `STATE` command from the player client, the Server should respond intelligently. Here, the player client is always “right” — the server registers the player as having aborted the match that the server was tracking, and then adjusts accordingly.

Figures X-X detail the handling of commands received by the Server, when a player is in each of the three middle states (`LONELY`, `PREGAME` and `GAME`).

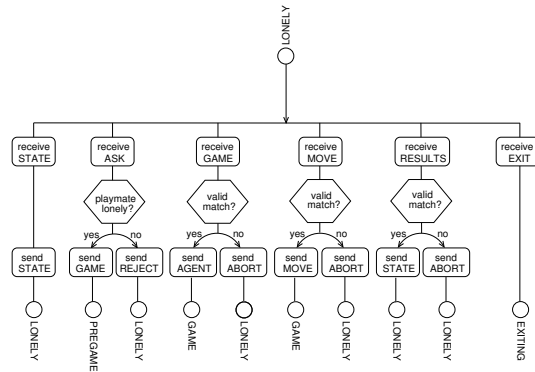


Figure X ServerLonely.

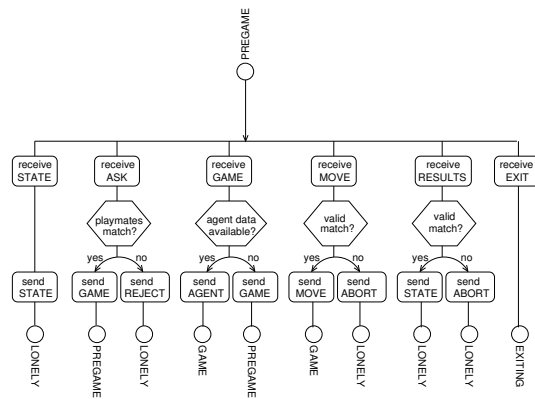


Figure X ServerPregame.

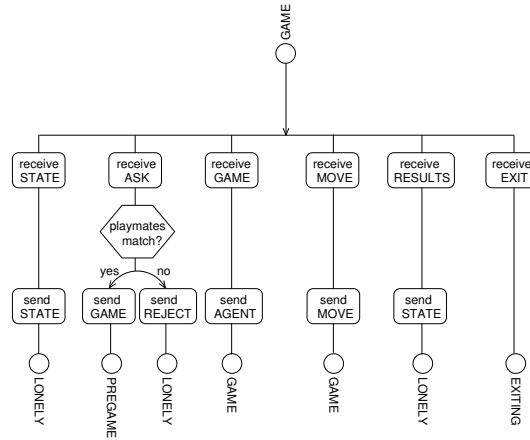


Figure X ServerGame.

## 2.9 Matchmaker.

this is a client that is used when a user connects to a game. the idea is that the system will attempt to see if any of the other users also connected to that game are appropriate “matches” to the current user, in order to keep students challenged by matches. however, i can’t remember how much of that idea was fully implemented. i think the matchmaker was the process that invokes the secret agents.

## 2.10 Client.

there are two types of clients.

(1) the playground client is written in C and compiled into a CGI. it uses HTML REFRESH to poll the server periodically to see what users are logged into the system.

(2) the game client is written in java. it maintains an open socket connection to the server while a game is going on.

this architecture was evolved because keeping a socket open for a long period of time was unreliable, when tested from the school’s computers. as long as a user was playing a game and there was two-way communication across the socket regularly, then that seemed to work okay. so we kept the java socket for the game, and replaced the polling through CGI for the playground mode.



### 2.10.1 Playground Client.

When a user first logs into CEL, the player is in an ENTERING state. After selecting a game to play, the player sends an ENTER command to the server, and the server puts the player in a LONELY state. This means that the user's browser is displaying a playground, as shown in figure 1a.

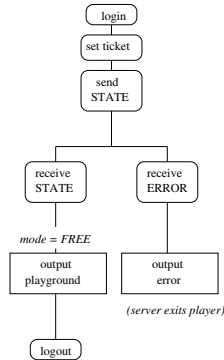


Figure X PlaygroundEnter.

The playground is an HTML page with a refresh command in its header. Every 5 seconds, the browser refreshes the playground page, which causes the player client to send a STATE command to the server, which in turn sends back the list of active playmates for that player. The player uses the list to draw an updated playground.

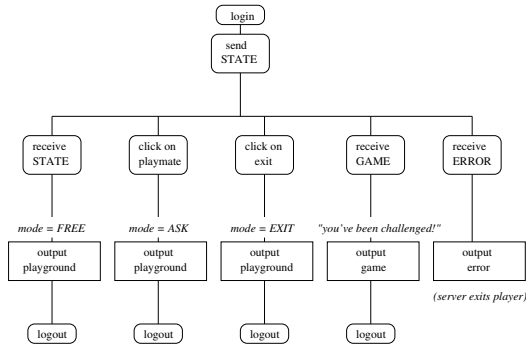


Figure X Playground.

If a user initiates a challenge, then an ASK command is sent from the player client to the server. If the server determines that the player is free to make a challenge and the player being challenged is also free, then the player client is placed in a PREGAME state. If the player is not free to make a challenge, then this means that another player has already challenged this player. In this case,

the earlier challenge takes precedence. The player is also placed in a **PREGAME** state, but the match is with the player who initiated the previous challenge. If the player who initiates a challenge is free, but her chosen opponent is not free, then the player receives a **REJECT** command from the server, and the player client remains in a **LONELY** state. Here, the player client also receives a current list of active playmates so her browser can update her playground.

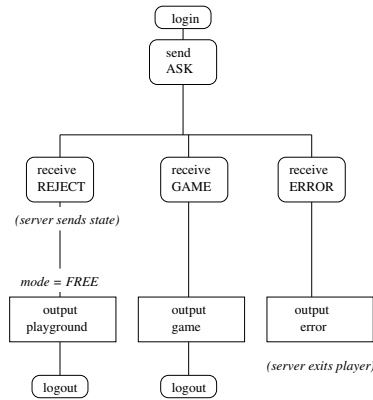


Figure X PlaygroundAsk.

A player can also be placed in a **PREGAME** state when her playground refreshes and a **STATE** command is sent from the player client to the server. At this time, if the server registers that a challenge has been accepted with this player, then instead of returning the player’s list of active playmates, the server returns a **GAME** command.

When a match is accepted at the server level and the initiating player is placed in a **PREGAME** state, the server sends a **GAME** command to the corresponding matchmaker (these terms are interchangeable — to avoid confusion, we will use the term “matchmaker” here). The matchmaker then fetches the data for the requested match and sends it back to the server in the form of an **AGENT** command.

A player client that is in a **PREGAME** state sends periodic (and frequent) **GAME** commands to the server, waiting for the game data to arrive. When the game data arrives at the server (i.e. the server has received an **AGENT** command from the matchmaker), the server forwards the **AGENT** command to the player client.

When a player wants to leave a playground, she clicks on an “exit” button which is displayed on every **CEL** page.

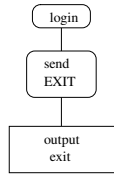


Figure X PlaygroundExit.

### 2.10.2 Game Client.

During the course of a game, the player client sends MOVE commands to the server. These are forwarded to the player's opponent. In this way, the game applets of the two players are synchronized.

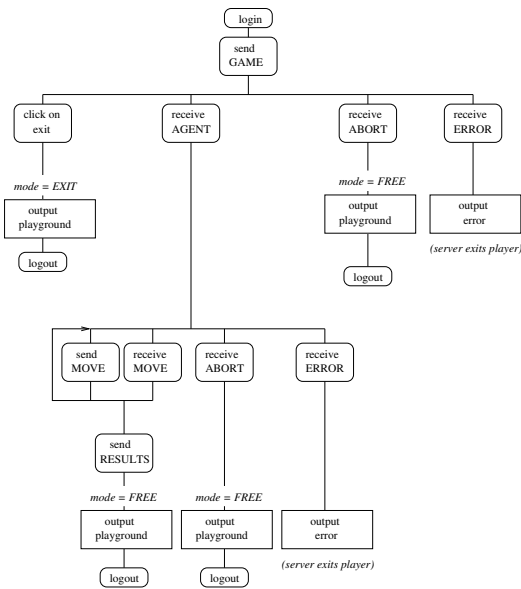


Figure X PlaygroundGame.

## 2.11 A Sample Game: *Keyit*.

### 2.11.1 Words database.

The content of word games in CEL is selected from a database of approximately 35,000 words. The dictionary file provided with a standard release of the Linux operating system is used. This file is called: /usr/dict/linux.words The version

provided with Linux 2.2.16. The file contains 45,402. Proper nouns and words deemed inappropriate for our audience (K-8th graders) were removed, resulting in our words file. A text version of our words files is called: CEL/data/words.txt. It contains 34828 words.

Every word is characterized by a set of seven features:

- word length,
- number of vowels,
- number of consonants,
- number of 2-letter consonant clusters,
- number of 3-letter consonant clusters,
- Scrabble<sup>®</sup> score, and
- keyboarding level.

The definition of these features is based merely on empirical evidence, with the exception of the keyboarding level. There is a standard order in which keys are introduced to students learning typing; the keys are presented in eleven groups. Thus each word in our dictionary is assigned a group number equal to the highest keyboarding level of any of the letters in that word.

### 2.11.2 Student Model

For each word that a student has been exposed to, an entry appears in the “word” portion of her student model, containing the number of times she has seen that word and her average typing speed for the word. Her overall average typing speed is also stored. When a match finishes, the game applet sends the results to the appropriate matchmaker, which appends any new words and updates existing ones and the overall average.

The student model is used as a basis for selecting game content. As shown in figure 5, we think of each word as a point in 7-dimensional space. Words that share feature values are close in this space; words with disparate feature values are further away.

In Keyit, the content for one game is considered a problem set consisting of 10 words. The first problem set (for a new user) is chosen at random. Subsequent problem sets are generated by comparing the result for each word in the most recently completed problem set with the student’s overall average. Words with a score that is better than the average are replaced with other words far away in the feature space (i.e. to *explore* new areas). Words with a score worse than the average are replaced with words that are nearby in the feature space (i.e. to *exploit* old areas).

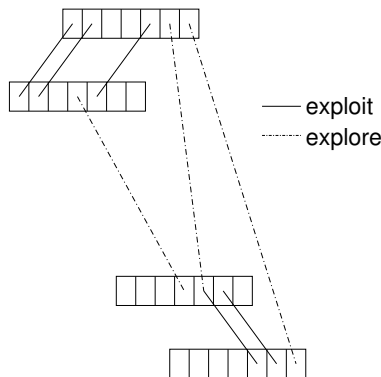


Figure 5: Problem set generation.

The student models are also used for choosing playgroup membership. As players connect to a game, the matchmaker fetches the corresponding user models and then computes a distance between all players currently logged into that game. A player's playgroup can then be restricted to contain only those mates who will provide appropriate matches. This is an important factor in controlling the competitive aspects of CEL.

## 2.12 System Architecture

The CEL system employs a modular client-server architecture, as shown in figure 2.1. One central server maintains a dynamic database indicating who is logged into the system and which games they are playing. This server also acts as a message passer, sending and receiving commands that go between clients.

There are six different types of clients in CEL: messenger, monitor, database manager, matchmaker, agent and player. The player client is designed to meet two fundamental needs:

1. to be practicable to anyone with Internet access and a web browser capable of running Java, and
2. to be usable by participants with limited network speed and low computer memory, as is the case for many school children.

As such, we use small footprint Java applets for games and implement the playgroups using CGI-bin programs that generate HTML and refresh periodically in order to update playgroup content.

The system is designed to be easily accessed by participants and easily extended by contributors (those adding their own games to CEL). Participants access CEL through the player client component, which runs inside their browsers. Section 2.19 highlights the characteristics of the player client that were built to meet the needs of school children. Contributors extend the matchmaker and

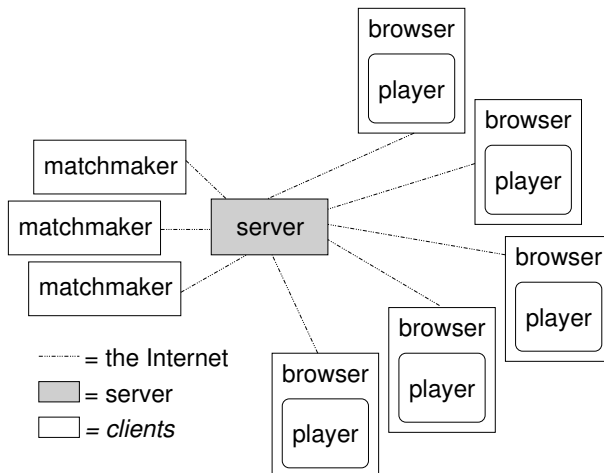


Figure 2.1: System Architecture.

agent components and the game portion of the player client component to implement their own activities in the system. Sections 2.17, 2.18 and 2.19, explain the details of each of these components, respectively.

This chapter describes each of the seven modules (one server and six types of client).

## 2.13 Server

The CEL server is a control component, having two primary functions:

1. to act as a central message processing facility, handling communication between all types of clients, and
2. to maintain a list of all the players who are currently logged into CEL and the status of each player.

The server is written in Java, version 1.0.2. We use Java version 1.0.2 because it can run inside Netscape version 3, which is (currently) more widely used than later versions of Netscape — supporting CEL's requirement for accessibility.<sup>3</sup>

The server interfaces with each of the six types of CEL clients (messenger, monitor, database manager, matchmaker, agent and player). The terminology

<sup>3</sup>Of course, we could use a later version of Java for our server and only restrict applet code to 1.0.2, but we decided it was simpler from a configuration management standpoint to use the same version for everything.

can be somewhat confusing because while CEL players may be thought of as general “clients”, they are not the only type of client. And while matchmakers may also be referred to as “game servers”, they are really clients as well. The distinction comes from network communication phraseology: the server opens a `ServerSocket` and each type of client opens a `Socket` in order to send and receive messages to and from the server (see figure 2.2).

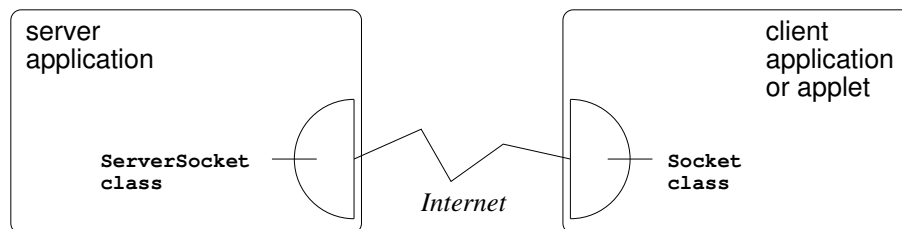


Figure 2.2: Sockets.

Commands are sent between the server and clients using the CEL command language (described in section 2.20). For example, messages are used to: log a client into and out of the system, register a player entering a playground or remove an exiting player, send a match invitation from one player to another, and pass game moves between players.

Figure 2.3 illustrates the `Server` and its relationship to each client application or applet. The components of the `Server` are shown above the solid grey line in the figure. The clients (shown below the solid grey line) are described in later sections of this chapter.

The `Server` extends the Java `Thread` class. It opens a `ServerSocket` on a specified port and listens for connections. When a new client makes a connection, the `Server` instantiates a `ServerClient Thread` to handle bi-directional communication with that client. The `Server` maintains a list of all active clients, i.e., a `Vector` of `ServerClients`.

### 2.13.1 Server Clients.

Each `ServerClient` extends a `Thread` and handles generic communication between the `Server` and any type of client. Its constructor initializes both input (`DataInputStream`) and output (`PrintStream`) streams for bidirectional communication on the socket. As long as the socket connection is alive, the client runs. It stops running either because it receives a command to stop (`LOGOUT`, `SHUTDOWN` or `KILL`) or because an exception occurs and the socket connection dies.

There are four flavors of `ServerClient` (see figure X):

1. `ServerMonitorClient`,
2. `ServerMatchmakerClient`,
3. `ServerPlayerClient`, and
4. `ServerAgentClient`.

When a client first connects, the `Server` instantiates a generic `ServerClient` to handle the connection temporarily. This is because when a client initiates a socket connection, it cannot pass any data to the `Server`, so the `Server` does not know which type of client is connecting. Once the connection has been established, the client sends a `LOGIN` command to the `Server`, which identifies the type of client (monitor, matchmaker, player or agent). Then the `Server` instantiates a class of the appropriate type (`ServerMonitorClient`, `ServerMatchmakerClient`, `ServerPlayerClient`, or `ServerAgentClient`) and places it in a `Vector` — either monitors, matchmakers or players (agents are stored in the `players` vector).

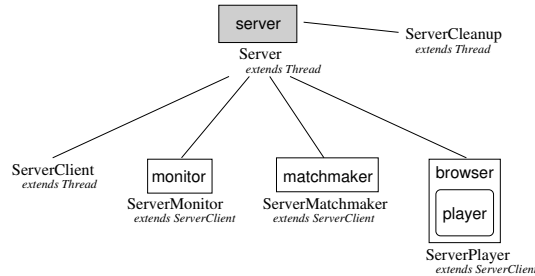


Figure X: .

As long as the socket connection with any `ServerClient` is alive, the server assumes that the client is running. When a client exits normally, it sends a `LOGOUT` command to the `Server` and closes the socket connection. Sometimes, the `Server` will initiate the closing of a client, either because the `Server` is shutting down, or because it has received a command to kill a particular client, or because the socket connection has died, which typically happens when a client exits abnormally.

### 2.13.2 Server Cleanup.

We cannot ensure that clients will exit CEL cleanly. Most clients are players that are instantiated in users' browsers. If a user clicks away to another web site or closes his browser, without logging out of CEL, then we have no way of knowing that the player has exited.

In the `Server`, it is important to keep the client vectors current. Of course, it is best not to use resources that are no longer needed, but more critical, it is necessary to know which players are alive and which have left CEL. This prevents the system from creating game matches that involve dead players.

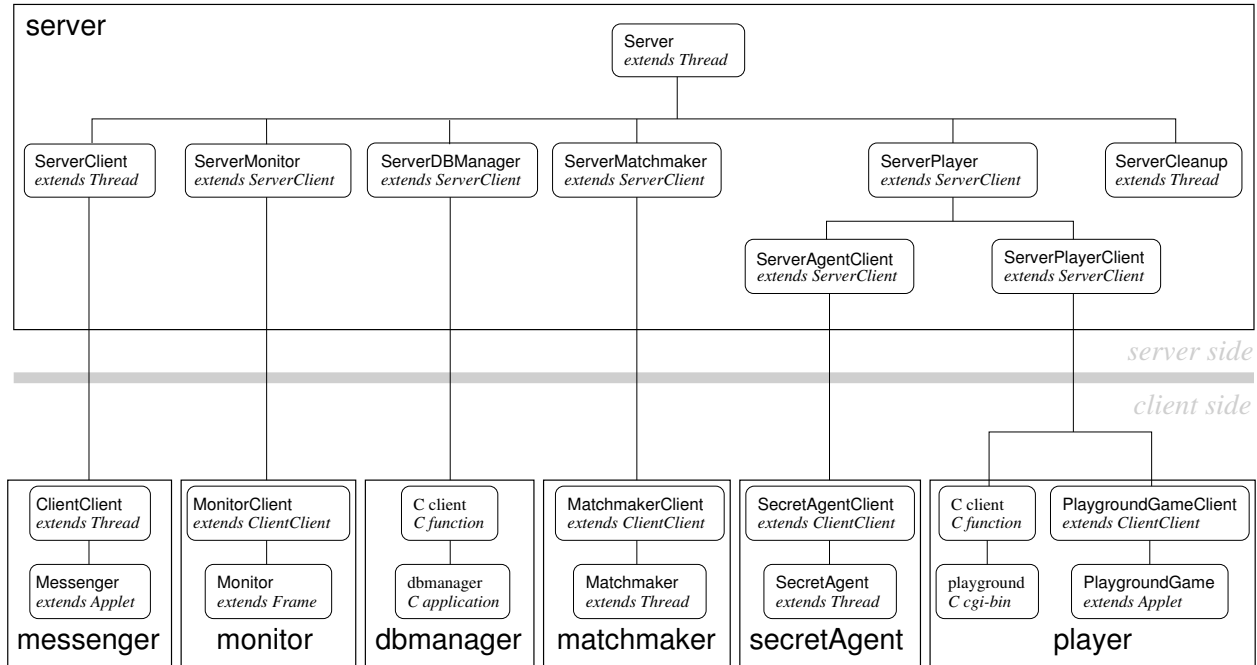


In order to maintain the integrity of the client vectors, the server instantiates a `ServerCleanup` thread. This thread runs periodically and clears out resources used by client threads that are no longer alive.

The `ServerCleanup` thread monitors the status of every client connection (player or otherwise), running periodically to check if any activity has occurred on each client's socket connection within a fixed time period. When it runs, it checks to see if any activity has occurred on each client's socket connection within a fixed time period. If no activity has occurred, then the `Server` sends a `PING` command to the client. The expected response is a `PONG` command, sent from the client back to the `Server`. If this is not received within a fixed time period, then the `Server` assumes that the client has disappeared and so the cleanup thread removes that client and all of its resources.

This process is necessary because we cannot ensure that clients (particularly players) will exit `CEL` cleanly. Players are instantiated in participants' browsers and if a user clicks away to another web site or closes his browser without logging out of `CEL`, then we have no way of knowing that the player has exited. So, in order to maintain the integrity of the active client list in the `Server`, we use the `ServerCleanup` thread. For example, this prevents the system from creating game matches that involve players who have left the system.

Figure 2.3: Server architecture, with overview of clients.



## 2.14 Messenger

The simplest client in CEL is the Messenger client. It is a Java application that provides a command-line interface for sending commands to the Server. It was built primarily as a development aid. The Messenger takes a message string in its command line and sends the message directly to the Server. The syntax of the message is the same as the CEL command language (see section 2.20).

## 2.15 Monitor

The Monitor is an expansion of the Messenger. It is also a Java application that provides a command-line interface for sending commands to the Server, but the Monitor also receives live feedback from the Server, reporting current status information on all active clients. The Monitor has a graphical front-end, which is pictured in figure 2.4. It can also run in a non-graphical mode, which is especially useful when testing CEL at a remote site where a graphics terminal is not available.

The Monitor can run on any networked computer, so it is a useful tool for contributors who are extending CEL.

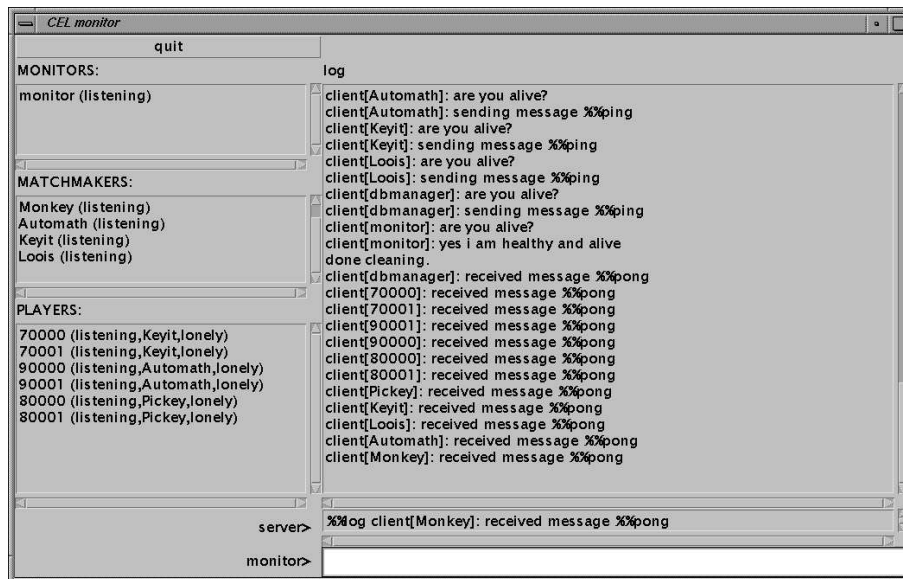


Figure 2.4: The Monitor.

## 2.16 Database Manager

The `dbmanager` is a C program that interfaces between the `Server` and the CEL databases. These include the student model component of the system as well as session logs.

The `dbmanager` runs as long as the `Server` is running. Whenever a player begins or ends a match, a message is sent from the `Server` to the `dbmanager`. The `dbmanager` parses the message and stores the relevant data in the appropriate CEL database table, as indicated in figure 2.5. Refer to chapter ?? for detailed information about the particulars of the CEL databases.

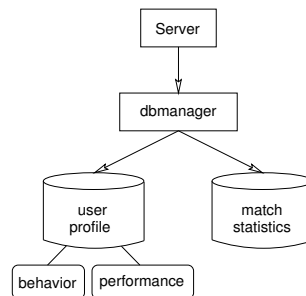


Figure 2.5: The `dbmanager`.

When a match begins, a record is written in the match statistics table, listing the name of the CEL game being played, the players involved in the match, and the start time of the match. As each player finishes the match, the table is updated with the end time and match result for that player. In addition, the user profile for that player is updated.

The user profile is divided into two components: behavioral data and performance data. The behavioral data indicates how frequently the user has played each game in CEL and whether the user is the initiator of each match. The performance data is domain-specific, rather than game-specific. For each domain element that a user has been exposed to, the time and result of each encounter is stored. Summary statistics are also maintained. This summarization is useful for computing similarity metrics between profiles for different users.

## 2.17 Matchmaker

For each game in CEL, there is one `Matchmaker`. This is a Java application that keeps track of all the users who are currently connected to its game. The `Matchmaker` is responsible for maintaining playgroups for each player, fetching

game content at the start of a match, instantiating agents to play the game when playgroups are too small and verifying moves during game play.

The components of the `Matchmaker` are shown in figure 2.6. The main class extends a `Thread`. Some of the components are highlighted in grey, to indicate that there may be multiple instantiations of these classes. The `MatchmakerClient` class extends the `ClientClient` class and handles bi-directional communication with the `Server`. A new `MatchmakerGame` class is instantiated every time a match begins. This is essentially a data structure that stores information pertinent to individual matches. When a match ends, its corresponding `MatchmakerGame` is removed.

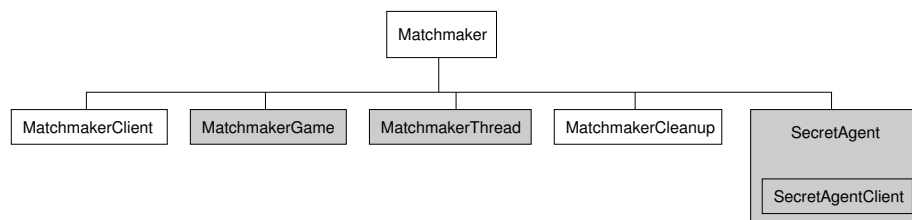


Figure 2.6: The Matchmaker.

The `MatchmakerThread` class can be used for one of two purposes: to fetch data for game content when a new match begins or to save data when a match is over. This thread is instantiated as either a `fetcher` or a `saver`. Both types invoke CGI-bin programs to interface, respectively, with the domain knowledge or the student model component of the CEL databases.

The algorithm used to fetch game content can vary, in order to maintain system flexibility. For example, formative evaluation of a system might involve comparing different methods for selecting content for the same game, which can be done simply by employing different CGI-bin programs to fetch the game data.

The `SecretAgent` extends the `Thread` class and runs as a child of the `Matchmaker`. Each `SecretAgent` instantiates its own `SecretAgentClient` class, an extension of the `ClientClient` class, to handle bi-directional communication with the `Server`. Once instantiated, a `SecretAgent` executes independently of its `Matchmaker` parent, although it dies when its parent dies. Secret agents can also be instantiated as programs and run autonomously, outside of a `Matchmaker`. This flexibility allows implementation of secret agents that have the ability to play multiple games. Detailed discussion of secret agents is found in section 2.18 and in chapter ??.

In order to implement their own activities, contributors extend the `Matchmaker`, `MatchmakerThread` and `SecretAgent` classes. For example, the `Keyit`

matchmaker is enabled through the `KeyitMatchmaker`, `KeyitMatchmakerThread` and `KeyitSecretAgent` classes. Matchmaker applications can execute on contributors' local machines as well as on our site. Instructions for extending classes to create a matchmaker can be found in [?].

## 2.18 Secret Agent

All the games in CEL are multi-player games. For the prototype implementation, these are all two-player activities. If not enough people are logged into a game playground, then it is useful to have software agents that can act as playmates. Otherwise, participants would be forced to wait until another human logs in before being able to play any games. We refer to the software agents in CEL as **Secret Agents**, because the agents may be indistinguishable from human playmates.

Figure 2.7 illustrates standard architecture for a software agent [?]. In CEL, the sensors and effectors are provided virtually by the `SecretAgentClient` class, which extends the `ClientClient` class and provides bi-directional communication with the `Server`. Through the `Server`, the `SecretAgentClient` receives information about the state of the world and sends its actions to be effected.

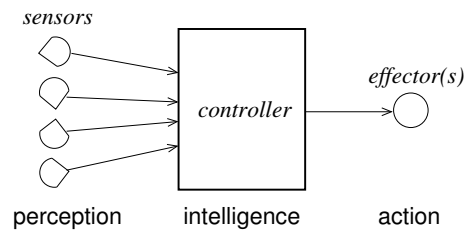


Figure 2.7: Typical software agent architecture.

The `SecretAgent` may perform three types of actions: system actions, playground actions and game actions. System actions refer to logging in and out of CEL. Playground actions refer to entering and exiting playgrounds and initiating challenges. Game actions refer to moves in a game.

The controller for a secret agent decides which action to take, given the input received from the `Server`. The flexibility of the CEL system makes it possible to design many types of controllers for secret agents. For the prototype version of CEL, we have defined two types of controllers. One is a simple reactive controller that does not initiate any challenges on the playground, and when playing a game, all its moves are direct responses to its playmates moves, based on a rule that allows the playmate to win almost every match, by a small margin. The second type of controller is a neural network controller that is trained to

emulate the behavior of humans who have visited CEL. Chapter ?? discusses the agents in more detail.

The `SecretAgent` class extends a Java `Thread`. It has one child: `SecretAgent-Client`. Secret agents can be instantiated as part of a `Matchmaker` (see section 2.17). They can also be run as independent programs. This allows more flexibility, particularly by permitting complex agents that exhibit system actions like logging in and out of CEL at particular times and being able to play different games.

Contributors need to extend the `SecretAgent` class in order to implement an agent that can participate in a new activity. Refer to [?] for instructions on how to extend this class.

## 2.19 Player

The CEL player client implements the user interface component of the system. It runs inside a web browser and has three elements: the menu, the playground and the game (see figure 2.8).

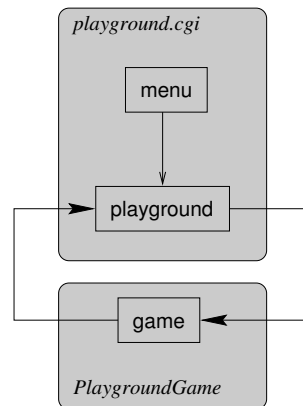


Figure 2.8: Overview of the player.

The menu and playground portions of the player client are implemented separately from the game portion. The following subsection details the development of the player module and adjustments made during formative assessment. The section concludes with a description of the final design.

### 2.19.1 Formative assessment

We performed formative assessment of the player client using the setup at a local primary school. The initial implementation of the player client was written entirely in Java. The playground was an applet, which opened a socket connection to the CEL server when it was launched and kept that connection open throughout a player’s entire session with CEL.

As illustrated in figure 2.9, the entire browser window was taken up with one playground applet. The background was an image, designed to look like the black-top in a school playground — white lines demarking hopscotch and a basketball court on a black background. Users’ IDsigns were inscribed in circles, whose color changed based on the state of each player. Players who were playing games were shown in green circles. Players who were sitting in the playground were drawn in white circles. A user’s own player was inscribed in a blue circle.

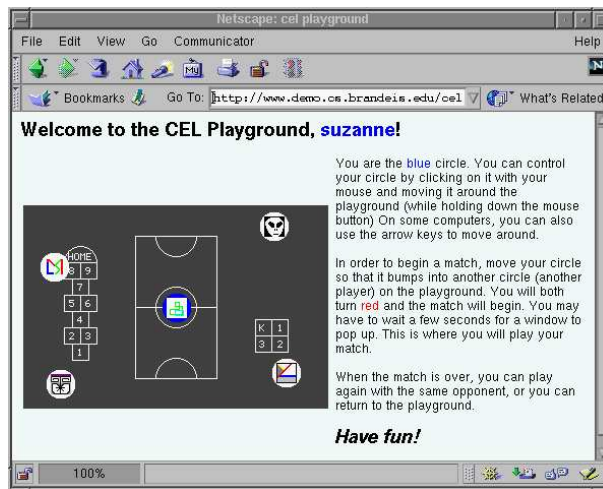


Figure 2.9: The CEL Playground, initial version.

The IDsigns were animated, moving around the playground in a fluid and dynamic manner. Each user controlled her own IDsign by clicking on it with her mouse. She could drag her IDsign and “bump” into that of a playmate; this action constituted an invitation to play a match. Playmates’ IDsigns moved in and out of the playground as users entered and exited the virtual space.

This design worked quite well in the laboratory, and children who tried this interface during one-to-one testing in our lab liked the style. Unfortunately, it proved too slow and cumbersome to be useful in a school setting. The amount of memory required for a large playground applet was too great for the computers



at our test site, a local primary school. Opening a socket to the server and keeping it open continuously did not prove to be reliable. The idea of moving IDsigns around with a mouse, while appealing to the children, did not perform well in practice, primarily due to limited memory.

We were fortunate to be able to work at a test site that is better equipped than most schools and may have faster access than many children do from home. But we wanted to make CEL accessible to school children across the U.S. and around the world, so the system had to perform to at least a median common denominator. If the performance was poor at our test site, we knew that it would not fare well in a typical school setting.

So, at first, we modified the design slightly, as shown in figure 2.10. Here the size of the applet was smaller (i.e., the amount of memory it used) because the image background was removed and the informational portion of the screen was done in HTML. Additionally, frames were introduced, so the left-hand portion of the screen, which was only HTML loaded up quickly and warned students to be patient while the right-hand frame (which contained the applet) loaded.

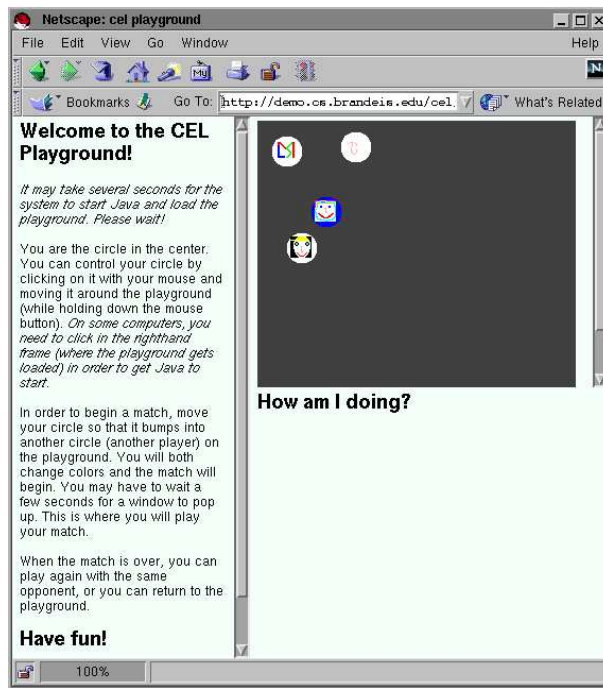


Figure 2.10: The CEL Playground, intermediate version.

We were disappointed to find that these alterations did not resolve the prob-

lems at our test site. The issues with memory were compounded by socket connection breakdowns. When the applet initialized, it opened a connection to the CEL server and attempted to keep that connection open during a user's entire session with CEL. However, we found that these connections kept getting interrupted. We tried implementing a recovery process whereby interrupted connections would reconnect to the server. But this resulted in more memory problems, because the memory allocated for lost connections was not recovered well in the browser and so as more connections were made, less and less memory became available on the students' computers and eventually they would hang.

As a result, we redesigned the player to use HTML and cue the browser to refresh the playground page periodically, asking the server for playgroup updates. This means that the player does not need to maintain a single long-term socket connection to the server. Conceptually, the playground is still an ongoing process that maintains a connection to the server so that the player can receive updates concerning who his current playmates are, as others enter and exit the site. In reality, the playground sends a refresh command to the browser telling it to update the playground every 5 seconds, thereby simulating a real-time connection to the server. Using this type of polling mechanism makes the system more accessible, because it accommodates clients with slow network connections and low memory computers.

The product of the redesign is a program called `playground.cgi`. At first, this was a shell script which invoked a Java application that connected to the Server, received an updated list of playmates and output HTML to draw the playground in the user's browser. After a few trials, we found that this method was also unacceptable. Every time the script was called, it started a Java virtual machine on our host computer in order to execute the Java application. This is bad because each Java virtual machine takes a while to start up, uses a lot of system resources while it runs and takes a while to close down and release the resources it was using — typically longer than the 5-second refresh period of each playground. When more than about 5 people were logged into CEL, our host machine was swamped. One player could have several `playground.cgi`'s running on our host because their browser would send a refresh command and start up a new one before the previous one(s) had completely exited the system.

### 2.19.2 Final design

The final `playground.cgi` is written in C and implements the menu and playground portion of the player client. This version runs in less than one second, and it works beautifully in practice. Although we gave up the fun gained from moving one's IDsign around in an animated environment, we are still able to emulate the dynamic nature of the environment and have ended up with a much more reliable and accessible product.

The game is a small-footprint Java applet, started when the `playground.cgi` outputs HTML containing an `APPLET` command. The applet is built on a class

called `PlaygroundGame`. When a `PlaygroundGame` applet initializes, it creates a child called `PlaygroundGameClient` that opens a socket to the `Server` and keeps that connection open as long as the game is in progress. When the game is over, the socket is closed and the applet invokes `playground.cgi` again, to fetch an updated playgroup from the `Server` and return to the state of waiting for a match. Again, accessibility is important, so the size of the applets are kept to a minimum, in order to lessen the memory requirements of clients' computers.

The user interfaces for individual games are enabled by extending the `PlaygroundGame` class. All of the games described in chapter ?? are built around this model. Contributors extend the `PlaygroundGame` class to implement the user interface for their own activities.

## 2.20 CEL Message Language

All communication between the `Server` and its clients is facilitated by the CEL Message Language. There are five classes of messages that are sent:

- I – commands between the server and any type of client
- II – commands between the server and messenger and monitor clients
- III – commands between the server and dbmanager client
- IV – commands between the server and matchmaker clients
- V – commands between the server and player and agent clients

Tables 2.1 through 2.5 list each set of commands, respectively, and describe their actions.

The message language was designed to support the extensibility requirements of the system. The separation between command classes protect the system; contributors who create matchmakers, players and agents cannot send destructive commands to the `Server`. For example, a matchmaker cannot effect a `KILL` command because it will not be recognized as a valid matchmaker command in the server.

The commands in classes III, IV and V were designed for flexibility, keeping short and simple the amount and type of communication that flows between the server and dbmanager, matchmakers, players and agents. The command set can support a variety of activities, as outlined in chapter ??.

Table 2.1: Class I: commands between server and any type of client.

<i>command</i>	<i>from server to client:</i>	<i>from client to server:</i>
LOGIN		client logs into CEL
LOGOUT		client logs out of CEL
PING	server checks socket connection	
PONG		client verifies socket connection
GETTIMEOUT	server reports timeout value for client	client receives value of timeout
SETTIMEOUT		server sets value of timeout
SEND	server sends a message to a client	client sends a message to another client, via server
ERROR		client reports that an error has occurred
SHUTDOWN	client shuts down	server shuts down ( <i>messenger and monitor clients only</i> )

Table 2.2: Class II: commands between server and messenger/monitor clients.

<i>command</i>	<i>from server to client:</i>	<i>from client to server:</i>
LOG		server writes a message to the log file
FLUSH		server flushes the log file
GETLOGINT	server reports value of log interval used during cleanup	client receives value of log interval used during cleanup
SETLOGINT		server sets value of log interval used during cleanup
GETCLNUPINT	server reports value of cleanup interval	client receives value of cleanup interval
SETCLNUPINT		server sets value of cleanup interval
CLEAN		server forces a cleanup to occur
WHO	server reports list of active clients	client receives list of active clients
KILL		server kills specified client

Table 2.3: Class III: Commands between server and dbmanager clients.

<i>command</i>	<i>from server to client:</i>	<i>from client to server:</i>
ENTER	server sends name of player entering a playground and time of entry	
EXIT	server sends name of player exiting a playground and time of exit	
RESULTS	server sends results of a match	

Table 2.4: Class IV: Commands between server and matchmaker clients.

<i>command</i>	<i>from server to client:</i>	<i>from client to server:</i>
GAME	server sends request for matchmaker to fetch game data	
DATA		matchmaker sends game data to server
RESULTS	server sends results of a match	

Table 2.5: Class V: Commands between server and player/agent clients.

<i>command</i>	<i>from server to player:</i>	<i>from player to server:</i>
ENTER		player enters a playground
EXIT		player exits a playground
STATE	server gets current state (list of active playmates)	player receives current state, with which to update playground
ASK		player requests match with specified playmate
GAME	match with requested playmate is accepted and server has requested game data (agent) from matchmaker	player has started game applet and requests game data (agent) from server
DATA	server sends game data (after data has arrived from matchmaker)	
MOVE	server passes move along to opponent	player sends its move to server
RESULTS		player sends results to server (and server passes them along to matchmaker)
REJECT	server rejects match with requested playmate	
ABORT	client aborts match, returns to lonely state	server forfeits match for client and returns client to lonely state
ERROR	client exits	server removes client

## 2.21 Player States

Both the **Server** and the **Player** keep track of a player's state. At any time, each player is in one of five states:

1. **ENTERING** — a new player is entering a playground
2. **LONELY** — a player is in the playground and is free to initiate a match or be invited by another player to engage in a match
3. **PREGAME** — a player has initiated a match or has been invited to play and is waiting for game data to arrive from the server
4. **GAME** — a player is playing a match
5. **EXITING** — a player is exiting a playground

Figure 2.11 summarizes the state transitions for players inside the **Server**, showing the normal flow between the five states.

All changes to the state are initiated by the **Player**. This is important. Sometimes the **Server** and a player can become out of synch. It is most critical for the actions in the player to seem logical to the human who is operating the player. So, all state changes are initiated by the player, and all discrepancies are resolved in favor of the player.

There are several ways in which the player and the **Server** can become inconsistent. Since the player is enabled inside a browser, a user could click on a navigation button to change the page and visit **CEL** pages out of sequence. For example, in the middle of a game, the user could use the “back” button in the browser to visit a playground that is several pages prior in the history list. In this case, the **Server** thinks that the user is in a **GAME** state, but upon receiving a **STATE** command from the player client, the **Server** should respond intelligently. *The player is always considered right* — so the **Server** registers the player as having aborted the match that the server was tracking, and then adjusts accordingly.

If response time is slow, users (especially children!) grow impatient. Sometimes they click on a **CEL** button multiple times or on the browser's reload button, while waiting for a response from the **Server**, which may result in an unexpected sequence of commands being sent to the **Server**. The mechanism described here is prepared to handle these types of discrepancies, which works to support the accessibility of the system, as **CEL** can operate robustly, responding reasonably to a variety of user behaviors.



## 2.22 Data in CEL

This chapter describes the data components of the CEL system, which include domain knowledge as well as data collected by the system as it runs. The domain knowledge is defined to be the information that the students are acquiring; in CEL, this is the content of games. We focus on three categories of on-line data collection in CEL (system products, session logs and survey results), demonstrating that CEL gathers the types of data required to support the kinds of activities common to the ILS field (see chapter ??).

## 2.23 Domain knowledge

Two types of domain knowledge databases have been defined for use by the CEL prototype activities:

1. a *words* database, and
2. an *arithmetic* database.

Keyit, Pickey and Monkey access the words database. Automath uses the arithmetic database.

The manner in which this data is stored and accessed is completely dependent on the learning activities that use the data. As indicated in chapter ??, the purpose of defining a domain database is to be able to identify elements within that database, in order to track a student's progress and (in CEL) to define game content. Some activities are designed to facilitate acquisition of a straightforward database of facts, e.g., multiplication tables, states and capitals, foreign language vocabulary. Database definition is easy in these cases because the domain can be broken down into individual elements and a one-to-one correspondence can be found between domain elements and elements of game content. For example, " $5 \times 6$ " can be defined as one element in a multiplication table database and this same equation can also be one (or part of a) problem to solve in a mathematics game. In the student model, the number of times a student has been asked to solve " $5 \times 6$ " can be tracked along with the number of times she has gotten the right answer, and then a simple numeric calculation can provide an indication of how well she has acquired that multiplication fact (e.g., percentage of correct responses).

Other domains are much harder to define. In the construction game Loois (section ??), it is difficult to enumerate the concepts that students should be acquiring. Here, the elements of game content are numbers and sizes of building blocks. But the intent is for students to acquire abstract skills such as elementary laws of physics, an understanding of torque, intuition about gravity, insights into structural integrity and experience translating ideas from simulation to reality. Keeping track of skill acquisition here is much more complicated and is

a research topic addressed by many working in intelligent tutoring systems. In future work with CEL, we plan to collaborate with others working in these areas to develop stronger methods for tracking student progress in complex domains.

The domain knowledge data sets that are used in the prototype version of CEL are described in the remainder of this section. The methodology defined should be taken as a suggestion for other possible domains and learning activities, but is not a requirement of the system.

### 2.23.1 *Words database*

The content of word games in CEL is selected from a database of approximately 35,000 words. Every word is characterized by a set of seven features:

1. word length,
2. Scrabble<sup>4</sup> score
3. keyboarding level,
4. number of vowels,
5. number of consonants,
6. number of 2-character consonant clusters and
7. number of 3-character consonant clusters.

The definition of Scrabble score is shown in table 2.6. For each word in the dictionary, the word's score is computed by adding together the scores for each character in the word. For example, the score for "millennium" is  $3 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 3 = 13$ .

The definition of keyboarding level is shown in table 2.7. There are several standards which define an order for introducing keys to students learning typing [?, ?, ?]; the latter was chosen arbitrarily. Each word in the dictionary is assigned a group number equal to the highest keyboarding level of any of the letters in that word.

---

<sup>4</sup>Scrabble is a word game that was invented in the U.S. by Alfred M. Butts in 1948 [?]. It is a board game in which players take turns making interconnecting words by placing letter tiles on a grid in crossword puzzle fashion. Each letter is assigned a fixed value, calculated according to its frequency of usage in everyday American English. Players receive a score for each word they place, calculated by summing the values for each letter in the word. In the 1950's and 60's, Scrabble gained popularity and spread to Canada, Great Britain and Australia. Today, the game is known all over the world.

Table 2.6: Scrabble scores.

character	score	character	score
A	1	N	1
B	3	O	1
C	3	P	3
D	2	Q	10
E	1	R	1
F	4	S	1
G	2	T	1
H	4	U	1
I	1	V	4
J	8	W	4
K	5	X	8
L	1	Y	4
M	3	Z	10

Table 2.7: Keyboarding levels.

level	keys introduced
1	a s d f j k l
2	e u
3	r i
4	g o
5	t h
6	w y
7	q p
8	c v
9	b m
10	x n
11	z

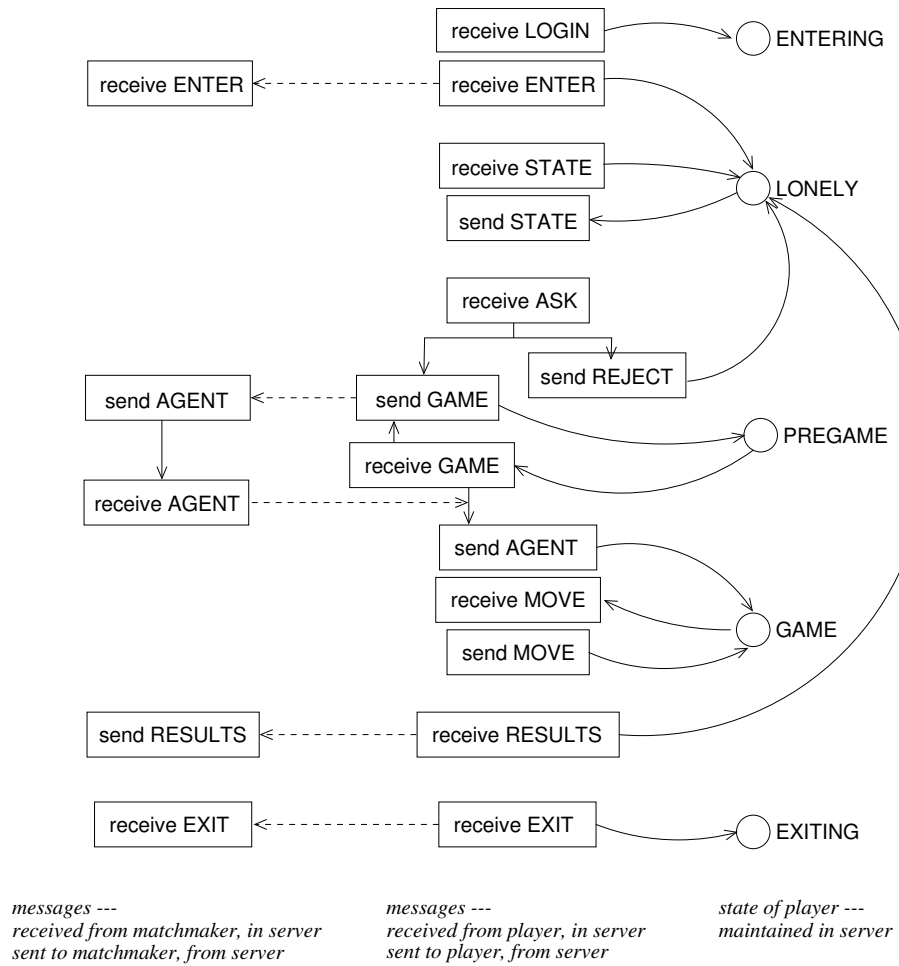


Figure 2.11: Player state diagram.

Figure 2.12 shows an example of the feature vector definition for the word “blue”. The reason for defining features for words is to provide a basis for selecting words as the content for word games. Some algorithms for selecting game content require that the domain be divided into a multi-dimensional feature space, so that data elements can be ordered in some fashion. For example, this allows the domain space to be partitioned into “easy” and “hard” segments.

Not all features will be relevant for all applications. For example, keyboarding level will not be relevant for spelling games. Experimentation may highlight which features are relevant for which applications (see chapter ??).

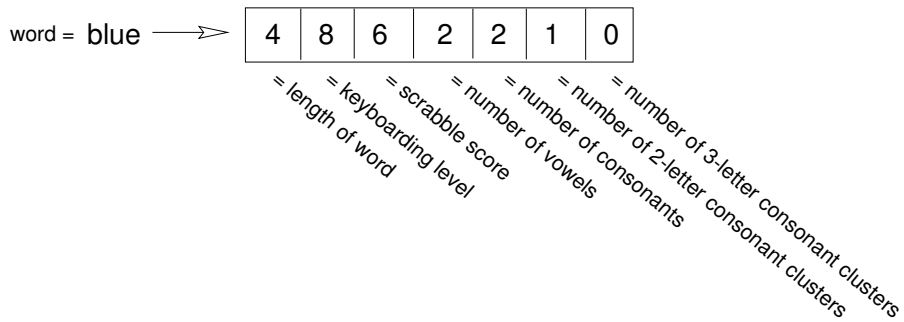


Figure 2.12: Sample word with feature vector.

### 2.23.2 Arithmetic database

For math games, the same level of traceability provided for word games is desirable — so that a student’s progress with a certain problem (or type of problem) can be tracked. Defining an element of knowledge in the arithmetic domain is more complicated than the method used for the words database. Is a formula the same as an element, e.g., “4 + 15”? Or is the answer to a problem the same as an element, e.g., “19”? The answer taken by itself seems too vague, but if it is a formula, then should “4 + 15” be the same element as “15 + 4”? What happens with complex problems, like “4 + 15 – (8 \* 7)<sup>2</sup>”? Is this one element or a combination of several?

Automath is the only currently operational math game, so the definitions used for this game will be presented here. An element is defined as follows for Automath:

`<sign_x><x><op><sign_y><y>`

where:

- `<sign_x>`, `<sign_y>` = positive (= 0) or negative (= 1)
- `<x>`, `<y>` = 0 . . . 1000
- `<op>` = addition (= 0),  
subtraction (= 1),  
multiplication (= 2),  
division (= 3),  
power (= 4)

To get a numeric identifier out of this, a binary string is constructed, by concatenating the numeric values for each of the five fields shown above.

Here is an example. The equation “2 – 5” is represented as follows:

<code>&lt;y&gt;</code>	<code>&lt;sign_y&gt;</code>	<code>&lt;x&gt;</code>	<code>&lt;sign_x&gt;</code>	<code>&lt;op&gt;</code>
5	+	2	+	–
000000101	0	0000000100	0	001

Concatenating the values in the third row results in the binary number:

0000000101000000001000001<sub>2</sub>

which equals

163873<sub>10</sub>,

and hence 163873 is the identifier for “2 – 5”.

This scheme does not allow for more complex formulae. For example, in the game of *Mathtree* (an arithmetic game that is currently being built), it is possible to have equations such as “4 + 15 – (8 \* 7)<sup>2</sup>”. It is desirable to use the same domain element definition for all math games (as with all word games), so

that a user's performance can be described with respect to the knowledge base and the task (i.e., a word and how to spell it) rather than the specific game. As more activities in mathematics domains are implemented, it may be necessary to alter our definition.

## 2.24 Data products

There are three categories of data collected in CEL: system products, session logs and survey results. The system products consist of student models and results of games. Some of the data is stored in a Postgres SQL database. Some data is stored in text and binary files. For the dynamic tasks that occur during system operation, access to the data must be quick, so standard disk files (text and binary) are employed. For post-operations analysis, it is more important that data be easily queried, so Postgres database tables are also employed. This section describes each type of data collected in CEL.

### 2.24.1 Student model

The student model has three components: demographics, behavior and performance data. Each is detailed in the following pages.

#### Demographics

User demographic data is stored in two tables in the Postgres database: `users` and `demographics`. Data is inserted into these tables when users log into CEL for the first time.

The `users` table contains the following information:

- `username` – a text value provided by users when logging into CEL for the first time.
- `password` – a text value provided by users when logging into CEL for the first time. The `password` is encrypted using the C function `crypt()`.
- `userid` – a numeric value assigned by selecting the maximum `userid` from the table and adding 1 to it.
- `consent` – a boolean value which is `true` when the user has clicked `okay` on a shrink-wrap consent form page, giving us permission to collect data on users' activities.

The `demographics` table contains the following information:

- userid – a numeric value that is the same as in the `users` table and can be used to cross-reference the two tables.
- gender – a character value that specifies the user’s gender (either “*m*” (male) or “*f*” (female)).
- age – an integer value specifying the age of the user.
- location – a text value that contains the country or state (if the country is U.S.) where the user is connecting from.
- address – a text value containing the user’s Internet (IP) address.
- language – a text field that specifies the user’s native language.

### Behavior

Behavior data is stored in the `behavior` table in the Postgres database. This table is written to by the `dbmanager`. Every time a user enters or exits a playground, initiates, finishes or forfeits a match, an entry is inserted into this table. Thus the data in this table can be accessed in order to study users’ behavior. For example, it is useful for training secret agents to emulate the playground behavior of humans.

The `behavior` table contains the following information:

- userid – a numeric value that is the same as in the `users` table and can be used to cross-reference the two tables.
- action – a text value that is one of the following:  
ENTER (when a player enters a playground)  
EXIT (when a player exits a playground)  
ASK (when a player initiates a match)  
RESULTS (when a player finishes a match)  
FORFEIT (when a player cancels a match)
- game – a text value indicating the name of the game in which the action has occurred (e.g., “Pickey”).
- timestamp – a numeric value specifying the time when the action occurred.

### 2.24.2 Performance

For each user, performance data is maintained both in a `rates` file and in the Postgres database. The `rates` files are updated and accessed by the `Matchmaker` applications. The database tables are updated by the `dbmanager` and accessed for post-operations analysis.

#### Rates files

The `rates` files are stored in binary form. One file is stored per user. Some rates files may be shared by multiple games, as is the case for `Keyit` and `Pickey`. `Monkey` has a separate rates file, as does `Automath`. The content of each file is:

`<nr><r0><r1> ... <rn-1><ng><g0><g1> ... <gng-1><avg>`



where:

- $n_r$  = integer, number of  $r_i$  records
- $r_i$  = a structure, of type RATE
- $n_g$  = integer, number of  $g_i$  values
- $g_i$  = integer, list of indices of domain elements in the last “generation” — i.e., the last problem set this user was given
- $avg$  = double, a game-dependent player performance average

This format is the same for all rates files. The definition of the RATE structure varies from one activity to another (see figures 2.13 and 2.14).

```
typedef struct {
    int    elemid;
    int    count;
    double sum;
} RATE;
```

Figure 2.13: RATE definition for Keyit, Pickey and Automath.

$n_g = 10$  and  $avg =$  the average score ( $sum/count$ ) for all  $r_i$  records. For each domain element: the number of times the user has encountered that element ( $count$ ) and the total of all scores for all encounters ( $sum$ ); thus an average rate for that element =  $sum/count$ .

```
typedef struct {
    int elemid;
    int count;
    int sum;
    int found;
} RATE;
```

Figure 2.14: RATE definition for Monkey.

$n_g = 1$  and  $avg =$  the average ( $sum/count$ ) for all  $r_i$  records. For each domain element: the total number of times this word was the monkey’s word ( $count$ ) and the total number of words found when this word was the monkey’s word ( $sum$ ); thus an average number of words found when this word was the monkey’s word =  $sum/count$  and the number of times the user has found this word ( $found$ ) inside the monkey’s word (see section ??).

## Rates tables

The *rates* tables are stored in the Postgres database. There are three inter-related tables, generally one set per activity, though games can share *rates* tables<sup>5</sup>. The tables have the name of the game which they belong to appended to the table name:

1. *rate\_game* (e.g., *rate\_Keyit*),
2. *rate\_x\_game* (e.g., *rate\_x\_Keyit*), and
3. *gen\_x\_game* (e.g., *gen\_x\_Keyit*).

The *userid* field is common to all three tables and is used to join them. It is the primary key for all the tables.

The *rate\_game* table is the master table. It contains one entry for each user (*userid*), which is akin to one *rates* file per user. The *rate\_game* table contains the following information:

- userid* – a numeric value that is the same as in the *users* table and can be used to cross-reference the two tables.
- average* – a real value akin to the *avg* field in the *rates* files.
- n\_rate* – an integer value akin to the  $n_r$  field in the *rates* files.
- n\_gen* – an integer value akin to the  $n_g$  field in the *rates* files.

The *rate\_x\_game* table contains one entry for each domain element that a user has been exposed to. It is akin to the  $r_i$  array in the *rates* files. For every record in the *rate\_game* table, there are *rate\_game.n\_rate* entries in the *rate\_x\_game* table, all with the same *userid*.

For example, the *rate\_Keyit* table contains the following information:

- userid* – a numeric value that is the same as in the *users* table and can be used to cross-reference the two tables.
- elemid* – an index pointing to a word in the *words* database.
- count* – an integer value indicating the number of times this user has been exposed to this word (*elemid*).
- sum* – a real value containing the sum of the user's scores with this word for all *count* times seeing this word.

The *gen\_x\_game* table contains one entry for each domain element in the last problem set the user attempted. It is akin to the  $g_i$  array in the *rates* files. For every record in the *rate\_game* table, there are *rate\_game.n\_gen* entries in the *gen\_x\_game* table, all with the same *userid*.

The *gen\_x\_game* table contains the following information:

---

<sup>5</sup>Pickey shares Keyit's tables.

- userid – a numeric value that is the same as in the `users` table and can be used to cross-reference the two tables.
- elemid – an index pointing to an element in the domain database.

### 2.24.3 Match Results

For each match played, results data is maintained in both a journal file and in the Postgres database. The journal file is updated by the `Matchmaker` applications and is accessed by the programs that report to users a record of their activities with each game. The database tables are updated by the `dbmanager` and are accessed for post-operations analysis.

The journal files are stored in text form. One file is stored per game. The format of the journal files is as follows:

```
<timestamp><client><userid><match><result>
```

where:

- <timestamp> = number of seconds since midnight on 01-Jan-1970
- <client> = the IP address of the player client
- <userid> = the player's userid number
- <match> = four-field entity that uniquely identifies every match
- <result> = result of the match, which varies for each game

The four-field `<match>` is defined as:

```
<timestamp><game><userid1><userid2>
```

where:

- <timestamp> = number of milliseconds since midnight on 01-Jan-1970
- <game> = name of game, e.g., `Keyit`
- <userid1> = userid number of player 1
- <userid2> = userid number of player 2

When a player finishes a match, a record is written to the journal file for that player, with that player's userid number as the third field in the record. This means that for matches where both players finish normally, there will be two records in the journal file for that match: one with player 1's userid in the third column and one with player 2's userid in the third column.

Note that the `<match>` field always lists the players in the same order, no matter whose results are contained in the record. Player 1 is defined to be the player who initiated the match. Player 2 is the player who accepted. If the match is between a human and a software agent, then the agent is always player 2.

#### 2.24.4 Survey Results

Whenever a user exits a playground, she has the option of completing a quick on-line survey and answering two questions:

1. how hard was the match?
2. how much did you enjoy the match?

Both questions are answered on a scale of 1-10. For the first question, 1 is defined to be “easy” and 10 is defined to be “hard”. For the second question, 1 is defined to be “boring” and 10 is defined to be “exciting!”.

Answers to this survey are stored in the Postgres database in the `opinion` table. Records are inserted into the table by a CGI-bin program which displays an HTML form with the two questions on it and processes users’ responses. The table is accessed for post-operations analysis.

The `opinion` table contains the following information:

<code>userid</code>	–	a numeric value that is the same as in the <code>users</code> table and can be used to cross-reference the two tables.
<code>timestamp</code>	–	a date value containing the time the survey was completed.
<code>game</code>	–	a text value that specifies the name of the game played.
<code>funness</code>	–	an integer value between 1 and 10.
<code>hardness</code>	–	an integer value between 1 and 10.

#### 2.24.5 System Logs

The system log file written in CEL maintains a chronological record of everything that happens while the system runs. This includes diagnostic information, which is kept for a month and then flushed.

The content of each log file is a record in the following format:

```
<timestamp><message>
```

The `<timestamp>` format is fixed (time in milliseconds since midnight, 01-Jan-1970). The `<message>` format varies. `[?]` contains detailed information on the content of the system logs. Most messages are formatted as follows:

```
client[<client>]: received message <message>
client[<client>]: sent message <message>
```

These tags indicate which client was involved in the communication.

Some examples are shown below (the timestamps were removed):

```
client[174]: received message %%pong
client[Pickey]: received message %%pong
```

```
client[null]: received message %%login 3 4 929645102
client[3]: received message %%ask Keyit 4
```

```

client[3]: sending message %%game 929645104412 Keyit 3 4

client[Keyit]: sending message %%game 929645104412 Keyit 3 4
client[Keyit]: received message %%agent 929645104412 Keyit %3 4 007 -31 10...
...11766 -1 embeds 9169 -1 curiousest 10667 -1 disinterestedness 22151 -1...

client[3]: received message %%move 929645104412 Keyit 3 4 3 11766 107.0 embeds
client[Keyit]: sending message %%move 929645104412 Keyit 3 4 3 11766 107.0 embeds
client[Keyit]: received message %%move 929645104412 Keyit 3 4 3 11766 107.0
client[3]: sending message %%move 929645104412 Keyit 3 4 3 11766 107.0
client[4]: sending message %%move 929645104412 Keyit 3 4 3 11766 107.0
client[3]: received message %%move 929645104412 Keyit 3 4 3 9169 225.0 curiousest

```

## 2.25 Summary

The data products collected in CEL may be accessed for many purposes:

- Student models may be accessed to select game content tailored to the needs of individual users. This is a dynamic task that occurs while the system is running and must contain current information that is consulted before every game a user plays.
- Student models may be used to define playgroup content. This is also a dynamic task that occurs while the system is running. Playgroups are updated after every game finishes and each time a new player enters or exits a playground.
- Match results are reported to users when requested. Playground pages show lists of all the matches a player has engaged in and the results. These lists update when a playground page is first loaded or dynamically if a user requests an update.
- User behavior and performance data may be used to train secret agents. See chapter ?? for an example.
- All types of data may be accessed for analysis, external to system operation.
- System logs are used to debug problems with the system.

The next chapter contains examples of the types of analysis that are typically performed in interactive learning systems, using the data products described here. We used many software tools (shell scripts, C programs, Matlab) in order to analyse the data. Future work involves building a high-level set of analysis tools that teachers can use to produce the same types of tables and graphs shown here.

## 2.26 CEL Databases.

There are three types of data being collected in CEL currently:

- user information
- user performance profile
- match results

### 2.26.1 User Information.

User information is stored in a Postgres SQL database. This is on *mopsus* and is called *cel*. To access it, on *mopsus*, type:

```
psql -d cel
```

If this doesn't work, make sure that the following line is in your *.cshrc* file:

```
source /homes/demo/sql/prepare
```

There are many tables defined in the *cel* database. The only one currently being used is called *passwords*:

Field	Type	Length
<i>user</i>	<code>varchar()</code> not null	25
<i>password</i>	<code>varchar()</code> not null	25
<i>id</i>	<code>int4</code>	4
<i>consent</i>	<code>bool</code>	1

The *user* and *password* fields are those provided by the user during their initial login session<sup>6</sup>. The *password* is encrypted using the C function `crypt()`. The *id* field is a number assigned by selecting the maximum *id* from the table and adding 1 to it. The *consent* field is `true` when the user has clicked `okay` on the shrink-wrap consent form page (this gives us permission to collect data on users' activities).

Soon, the DEMO login facility will begin collecting additional optional information from users. This data will go in the *demographics* table:

---

<sup>6</sup>Note that *user* needs to be changed to *username* in order to upgrade to the latest version of Postgres (in which *user* is now a keyword).

Field	Type	Length
id	int4 not null	4
gender	char()	1
age	int4	4
location	varchar()	30
address	varchar()	50
language	varchar()	10

The `id` field is the same as in the `passwords` table and can be used to cross-reference the two tables. The `gender` field is either “*m*” (male) or “*f*” (female) and specifies the user’s gender. The `age` field specifies the age of the user. The `location` field contains the country or state (if the country is U.S.) where the user is connecting from. The `address` field contains the user’s IP (Internet protocol) address. The `language` field specifies the user’s native language. Note that this login facility is also used by DEMO’s Tron and Shock projects, and eventually (hopefully!) by the Backgammon project.

### 2.26.2 User Performance Profile.

For each user, a performance profile is maintained. These are stored on *mopsus*, in binary files named:

`/home/httpd/html/cel/data/rates/<id>`

where `id` is the user’s id number (assigned in the login procedure above).

The format of each rates file is:

`<nr><r0><r1> ... <rn-1><ng><g0><g1> ... <gn-1><avg>`

where:

`nr` = integer, number of `ri` records  
`ri` = record, of RATE type defined in `rate.h` as:

```
typedef struct {
    int bitbot;
    int count;
    double sum;
} RATE;
```

This indicates, for each `bitbot`, the number of times the user has encountered that `bitbot` (`count`) and the total of all scores for all encounters (`sum`) — thus an average rate for that `bitbot` = `sum/count`.

`ng` = integer, number of `gi` values  
`gi` = integer, list of `bitbots` in the last “generation” — i.e. the last problem set this user was given  
`avg` = double, the average score (`sum/count`) for all `ri` records

What is a `bitbot`? A `bitbot` is an integer that is assigned to identify individual “bits” of information in each knowledge base that CEL taps for game content.

For example, for the word games, there is a single database of 35,000 words. Each word is assigned a unique integer — a *bitbot* — which serves as an index for that word.

This works well for the word games. But for the math games, it may not be as good. We want to have the same level of traceability, so we know if a student has been exposed to each type of problem. But what is a “bit” of knowledge in a math game? Is it a formula, e.g.  $4 + 15$ ? Is it a number, e.g. 19? A number by itself seems too vague, but if it is a formula, then should  $4 + 15$  be the same bitbot as  $15 + 4$ ? What about for more complex problems, like  $4 + 15 - (8 * 7)^2$ ? Is this one bitbot or a combination of several?

*Automath* is the only currently operational math game. A bitbot is defined as follows for *Automath*:

$\langle \text{sign}_x \rangle \langle x \rangle \langle \text{op} \rangle \langle \text{sign}_y \rangle \langle y \rangle$

where:

$\langle \text{sign}_x \rangle, \langle \text{sign}_y \rangle$  = positive (= 0) or negative (= 1)  
 $\langle x \rangle, \langle y \rangle$  = 0 . . . 1000  
 $\langle \text{op} \rangle$  = addition (= 0),  
 subtraction (= 1),  
 multiplication (= 2),  
 division (= 3),  
 power (= 4)

To get a numeric bitbot out of this, we encode a “chromosome” as a binary string made up by concatenating the numeric values for each of the five fields shown above, as shown below.

For example,  $2 - 5$  is encoded as:

$\langle y \rangle$	$\langle \text{sign}_y \rangle$	$\langle x \rangle$	$\langle \text{sign}_x \rangle$	$\langle \text{op} \rangle$
5	+	2	+	-
000000101	0	0000000100	0	001

which becomes the binary number: 0000000101000000001000001  
 which equals  $163873_{10}$ , and hence is the bitbot for  $2 - 5$ .

This does not allow for more complex formulae. But likely, such will be needed for *Mathtree*. So the bitbot definition is an open question for the math games. We want to use the same structure for all math games (that is, arithmetic games), just as for all word games so a user’s performance profile is defined with respect to the knowledge base, not the specific game.

### 2.26.3 Match Results.

For each match played, entries are made in an ASCII file. There is one such file for each type of game in *CEL*. These are stored on *mopsus* and are named:



/home/httpd/html/cel/data/<game>.all  
where *game* is the name of the CEL game, e.g. *Keyit*. These are referred to as all files. For each match played, there are one or two lines in the all file.

The format of the all files is as follows:  
<timestamp><client><id><match><result>  
where:  
<timestamp> = number of seconds since midnight on 01-Jan-1970  
<client> = the IP address of the player client  
<id> = the player's id number  
<match> = four field entity that uniquely identifies every match  
<result> = result of the match, which varies for each game

The four field <match> is defined as:  
<timestamp><game><id1><id2>  
where:  
<timestamp> = number of milliseconds since midnight on 01-Jan-1970  
<game> = name of game, e.g. *Keyit*  
<id1> = id number of player 1  
<id2> = id number of player 2

When a player finishes a match, a record is written to the all file for that player, with that player's id number as the third field in the record. This means that for matches where both players finish normally, there will be two records in the all file for that match: one with player 1's id in the third column and one with player 2's id in the third column.

Note that the <match> field always lists the players in the same order, no matter whose results are contained in the record. Player 1 is defined to be the player who initiated the challenge. Player 2 is the player who accepted. If the match is between a human and a software agent, then the agent is always player 2.

Here are two examples from *Keyit.all*. First, a normal completion for player with id #3:

```
939236921 129.64.3.248 3 939236880313 Keyit 3 70003 107.6 30517 93 8408 144 28841 97  
20308 159 5781 67 34286 141 6578 67 34712 148 28549 68 25808 92
```

In this case, the <results> are as follows: average score for the match, which is the average typing speed (per word) in hundredths of a second, followed by pairs of numbers — the first number is the *bitbot* (id number) and the second number is the typing speed, in hundredths of a second.

Second, a forfeit for player with id #3:

939236732 129.64.3.248 3 939236700453 Keyit 3 70005 forfeit

In this case, the <results> are that player #3 forfeited the match.

## 2.27 CEL Log files.

There are three types of log files being written in CEL:

- server
- matchmaker
- agent

### 2.27.1 Server Log Files.

The server log files are written to *mopsus* and are named:

`/home/httpd/html/cel/logs/<server-id>.<DDMMYYYY>.log<.timestamp>`

Notice that CEL is Y2K compliant.

The educational CEL server id was previously *server*, but has just been changed it to *Educel*, to distinguish from server logs written by the *Tron* and *Shock* communities. The <.timestamp> portion of the file name is optional.

Every time a server starts up, it first looks to see if a log file with the specified name <server-id>.<DDMMYYYY>.log exists. If it does, then the server renames that file, appending the <.timestamp> and then opens a new log file (without the prefix).

The content of each log file is a record in the following format:

<date><message>

For example:

Mon Mar 29 17:24:12 EST 1999 log file opened a-okay

The <date> format is fixed. The <message> message format varies.

In order to save some space in the log files, the <date> format has recently been changed to <timestamp>, which is in milliseconds (since midnight, 01-Jan-1970). This field a single string.

The possible messages are listed below.

```

cleaning...
cleanup : deleting dead player <id>
cleanup : deleting non – playing player <id>
cleanup : deleting non – refreshing player <id>
cleanup : monitors = <list> | matchmakers = <list> | players = <list> |
client [<game>] : in finishMatch(), match not found (<match>)
client [<id>] : are you alive?
client [<id>] : connection is dead
client [<id>] : in game(), invalid match : <match>
client [<id>] : in move(), bad match opponent
client [<id>] : java.net.SocketException : Broken pipe
client [<id>] : received message %%abort <match>
client [<id>] : received message %%ask <game> <id2>
client [<id>] : received message %%enter <game>
client [<id>] : received message %%game <match>
client [<id>] : received message %%logout
client [<id>] : received message %%move <match> <move>
client [<id>] : received message %%pong
client [<id>] : received message %%results <id> <match> <result>
client [<id>] : received message %%state <game>
client [<id>] : sending message %%agent <match> <agent>
client [<id>] : sending message %%game <match>
client [<id>] : sending message %%move <match> <move>
client [<id>] : sending message %%ping
client [<id>] : sending message %%reject invalid match %%state <mates>
client [<id>] : sending message %%reject the playmate you picked is busy – please try again! %%state <ma
client [<id>] : sending message %%reject your playmate is not in the <game> playground %%state <mates>
client [<id>] : sending message %%sshhuuttdoowwnn
client [<id>] : sending message %%state <mates>
client [<id>] : yes i am healthy and alive
client [<game>] : are you alive?
client [<game>] : received message %%agent <match> <agent>
client [<game>] : received message %%pong
client [<game>] : sending message %%bond
client [<game>] : sending message %%game <match>
client [<game>] : sending message %%ping
client [<game>] : sending message %%results <id> <match> <result>
client [<game>] : sending message %%sshhuuttdoowwnn
client [<game>] : yes i am healthy and alive

```

