Tim Henderson

Steve Johnson

Leigh Praskac

**Diplomacloud**

The original game of international intrigue taken to the cloud

February 12, 2009

Progress Report Part 1

**1. Introduction: Overview**

Diplomacloud will provide an online, computer-judged form of the original Diplomacy board game. Unlike similar strategy games such as Risk, Diplomacy is a game determined by skill and inter-player alliances, *not luck*. Seven players compete for the conquest of a map divided into territories. Each player represents one country. During negotiations, players write orders for each of their units and submit them to the board master. When all of the orders have been turned in, the board master reads all of the orders and pieces are moved. Conflicts are resolved using a rule set, which is attached to this report. This order-and-resolution mechanism makes Diplomacy an ideal game for the online database format required for this project. Players will create games, view the current game map, and submit orders online. The program handles conflict resolution and map views automatically.

The basic system is already implemented on a Linux server (masran.case.edu/diplomacy/main.py). The server is running the Apache2 web server to interface with the web. The application is written in the Python 2.5 programming language. The database management system used is MySQL. Python interfaces with MySQL using the MySQLdb API python module. Python interfaces with Apache2 using CGI. Apache is configured to execute python on the specified scripts, rather than using mod\_python.

All HTML pages in Diplomacy will be dynamically generated. A templating engine called YAPTU (Yet Another Python Templating Utility) is used to create the HTML templates. Each web page seen by the user will have two parts on the back end: the HTML template, and the python script that interfaces with the database and generates the page based on the template.

Most of the queries run against the database will be in the form of MySQL stored procedures, which allow the programmers to create a logical separation between database interaction and the game logic. However, MySQL’s implementation of stored procedures is not entirely adequate for all situations. For instance, stored procedures do not provide an efficient method for inserting large datasets into the database. In these instances, dynamic SQL will be generated by python and executed on the database.

To protect our database from being compromised, we will employ several measures. The first will be a special user account with very limited permissions just for the application. Next, each web-submitted value that needs to be captured by the database will be validated and escaped to ensure that it does not result in a SQL injection. Using stored procedures also enhances security by having most of our queries parameterized. Although this method is not fool proof, by carefully auditing our procedures we can ensure that they are safe from injection. Finally, our application already has in place an extensive logging capability so we can capture auditing and debugging information in text files. These data will not be captured in the database, so if database fails, the operators will be able to find out why.

**2. Application Requirements Specifications**

Every game play webpage will include several items: the standard header, footer, and navigation bar in addition to a map showing current unit positions, a season/year counter, and a countdown clock to order submission deadline. The game will be played on a map generated by a random procedural algorithm that we have already written. Players will start the game owning a small number of territories determined by the algorithm. An example map is shown in the attached images. To avoid problems with text label overlap, territory names will be abbreviated to three-letter acronyms. A random name generator will provide a long name and an acronym for each territory.

The game map is arguably the most important part of the game play interface. It shows units as colored shapes representing country (color) and type of unit (army or fleet). The season/year counter displays the current turn of the game. Turns represent six-month periods that rotate through the spring and fall seasons. To be more explicit, games start in the spring of 1901, then progress into the fall of 1901, the spring of 1902, the fall of 1902, and so forth. Orders for each season must be submitted within a given period of time, as kept track of by the countdown clock. Once the clock expires, order resolutions are processed and the season advances.

Each season consists of four stages: 1) movement order writing 2) order resolution 3) retreat/disbandment order writing 4) retreat/disbandment order resolution. Stages 1 and 3 require separate pages. Stages 2 and 4 will be executed internally. Fall seasons include a fifth and sixth stage: 5) gaining/losing units order writing 6) gaining/losing units order resolution. Stage 5 requires a page; stage 6 will be executed internally. Two additional pages display current supply center count for all countries and the final map at the end of the game. In the following paragraphs, the requirements for each page type will be discussed. Example pages of each type are attached to this report.

Games will be started by single users who will add other users to the game. The game creator will choose a map from a paged list of named thumbnails, set a world name, and specify a maximum turn length. When the game creator is ready, the server will randomly assign countries to players and start the game clock.

(Stage 1) Movement Order Page

For each unit of a given country, four types of orders are possible: Hold, Move, Support, and Convoy. Players give orders to each piece individually. If no order is specified, then the unit defaults to Hold. This page therefore needs to provide a way for the player to specify orders for each of their units.

For the player logged in, a list of their units appears next to the current game map. Next to each unit is the specified order for that unit (all units start at Hold). Clicking on the unit name brings an option to change the order. Move orders specify where the unit should move to; Support orders specify which unit to support; and Convoy orders specify which unit to convoy.

When the player is finished specifying orders, the save button is pressed and orders are saved to the database. Players may modify their moves at any time before the countdown clock expires. Once the clock expires, the most recently saved order set is sent for order resolution. If the player did not save orders before the clock expired, all units default to hold.

(Stage 3) Retreat/disbandment Order Page

When an enemy succeeds in taking over a territory, the defeated unit must vacate the territory. This defeated unit has two options: Retreat or Disband. The default order is to disband. This page, then, provides a way for players to specify Retreat/Disband orders for their defeated units.

Similar to the Stage 3 Page, this page displays all of the player’s defeated units next to the current game map. Next to each unit is the specified order for that unit (all units start at Disband). Clicking on the unit name brings up an option to change the order. Retreat orders specify which territory to retreat to. Note that players without defeated units are shown a page displaying the current map only.

Orders are processed in the same way described in the Stage 3 Page section.

(Stage 5) Gaining/Losing Units Order Page

At the end of each Fall turn, supply centers are reassigned. Countries with supply center counts greater than the number of units they currently have on the board are allowed to build new units on their home centers, while countries with supply center counts less than the number of units they currently have on the board must disband units. This page provides a way for players to specify Build and Disband orders.

This page shows either the number of units a player is allowed to build or the number of units a player must disband. If building is the option, the player selects which home center(s) will build what unit type(s). The default is to build no units. If disbanding units is an option, the player selects which units to disband. In the case of no selection, a random unit is selected for disbandment (the unit is selected from among units non-home territories if possible). Note that players not allowed to build or disband are shown a page displaying the current map only.

Orders are processed in the same way described in the Stage 3 Page section.

Supply Center Count Page

This page is common to all players. It consists of a table of all supply centers on the map listed with the current owner of that territory. A second table shows the current supply center count of each country. No user interaction is involved.

End of Game- Map View (and statistics?) Page

When a game is finished, the final map will be displayed on this page. No user interaction is involved.

**3. Database Requirement Specification**

**Entities and Relationships**

Most of the model is represented in a partial entity-relationship diagram, attached to this report. We made the diagram using an automated system called GraphViz so that we could simply specify entities and relationships in text without having to draw arrows and rearrange graphical elements. The generated arrangement is not perfect, but it is moderately easy to understand with the accompanying notes.

We have already set up a server to work with and created all of the tables. The MySQL code to create the tables is attached to this document. The code should clear up any ambiguities within the explanations.

*Game(gam\_id:int, map\_id:int,pic: string, season:int, gameyear:year, turn\_start:datetime, turn\_length:time, turn\_stage:int, ended:int)*

*Game* contains an auto-incrementing ID, a map, and its current season and year. *Game* does not relate directly to *User* because each user can only play one country per game, and to relate *Game* to *User* would introduce redundancy. To get a list of users, we can simply look at *Countries*.

The *Game-Uses-Map* relation is implemented with a simple *map\_id* foreign key, since a game can have only one map. Since the same map is shown to all players in a game but different forms of the same map are shown to different players in different games, the *Game* entity also has a *pic* attribute, which is the name of the image file that will be shown to the players.

The current state of the game is preserved in the *season* and *gameyear* attributes. The next-turn countdown information is stored in *turn\_start* and *turn­­\_length*. The *turn\_start* attribute is set to the current time whenever a turn begins, and *turn­\_length* determines the maximum amount of time a turn can take.

There are multiple stages in a turn, as explained in the previous section. The online interface will display different options based on the current stage. The *turn\_stage* attribute keeps track of the current turn stage. It is a foreign key to a *TurnStage* entity, which is a simple integer-string pair.

*Map(map\_id:int, world\_name:string, pic:string)*

*Map* contains basic data about the game world. *World\_name* is simply the map’s given name so that users can distinguish between maps they have already played. *Pic* is the name of an image file (minus the extension) containing the currently shown version of the map.

The *Map-Has-Country* relation is implemented with a *map\_id* foreign key in *Countries*.

*Territory(ter\_id:int map\_id:int, name:string, abbrev:string, piece\_x:int, piece\_y:int, label\_x:int, label\_y:int, supply:int, coastal:int, ter\_type:int)*

Territories belong to *Map* and have a full name and abbreviation. The attributes *label\_x* and *label\_y* are used to display the name in the correct position. To display a piece that occupies the territory *piece\_x* and *piece\_y* are used. The *map\_*id foreign key implements the *Map-Has-Territory* relation. The *supply* attribute determines whether the territory is a supply center, and *coastal* is set if the territory is land and borders a sea. The *ter\_type* attribute determines whether the territory is land or sea.

*Adjacent(ter\_id:int, adj\_ter\_id:int)*

Territory adjacencies make up a many-to-many relationship, so an extra table is necessary to store adjacencies. Both attributes are foreign keys to *Territory*, and the primary key of *Adjacent* is the pair of both keys, since no adjacency pair should ever appear in the table twice. Adjacencies will be stored both ways (a-b and b-a) to make queries simpler and faster.

*Line(ln\_id:int x1:int, y1:int, x2:int, y2:int)*

Visual map data must be stored in the database, which means that we have to define a number of primitives. The *Line* entity represents a simple graphical line. Lines can be part of one or more territories, so the *Line-In-Territory* relation is not implemented in the *Line* entity itself.

*LineInTerritory(ter\_id:int, ln\_id:int)*

Lines can be part of more than one territory, since a line determines the boundary between two territories. As such, the *LineInTerritory* entity simply has foreign keys to *Territory* and *Line*. The pair of both forms the primary key.

*Triangle(tri\_id:int, ter\_id:int, x1:int, y1:int, x2:int, y2:int, x3:int, y3:int)*

Territories are graphically composed of one or more triangles Triangles are always part of exactly one territory, so the *Triangle-In-Land* relation is implemented with the *ter\_id* foreign key to *Territory*.

*Country(cty\_id:int, usr\_id:string, map\_id:int, name:string, color:string)*

Each *country* is controlled by a single user and exists on a single map. Players name their own countries. Colors are specified in 7-byte hex strings, e.g. “#ff0000” for opaque red. The *User-Controls-Country* relation is implemented with a foreign key *usr\_id*. The *Map-Has-Country* relation is implemented with the *map\_id* foreign key to *Map*.

*GamePiece(pce\_id:int, cty\_id:int, ter\_id:int, pce\_type:int)*

Game pieces belong to individual countries, only one at a time. They can also occupy only one territory at a time. A game piece can be either a fleet or an army, hence the *type* field. The *Country-Has-GamePiece* relation and the *GamePiece-Occupies-Territory* relation are implemented with foreign keys.

*Supplier(ter\_id:int, cty\_id:int)*

Territories are assigned supplier status at the map generation stage, but supply centers are owned by different countries in different games depending on how the game plays out. To deal with multiple concurrent games being played on the same map, we need a separate *Supplier* entity to keep track of the current game state. *Supplier* contains foreign keys to *Territory* and *Country*, and the pair forms the primary key. Multiple games reference the same territories, territories can supply one country per game, and multiple games can run at the same time, so neither foreign key alone is sufficient to be a primary key for *Supplier*. However, the pair is sufficient, since a territory can only supply one country at a time in one game.

The foreign keys implement the *Supplier-Represents-Territory* and *Supplier-BelongsTo-Country* relations.

*Order(ord\_id:int, cty\_id:int, pce\_id:int, season:int, gameyear:year, order\_type:int, destination:int, executed:int)*

Orders are the most important part of the game play, and they are somewhat sophisticated. A simple order operates on a single piece, specified by the foreign key *pce\_id*. The *pce\_id* attribute implements the *Order-Commands-GamePiece* relation. Orders also have a season and year, allowing for a page showing a summary of past orders. The *order\_type* attribute specifies which order is being given, e.g. hold, attack, etc. It is a foreign key to an *OrderType* table that simply stores type-string pairs. Almost every order has exactly one destination with the exception of Hold and Move-Via-Convoy, so there is a *destination* attribute for convenience. The *executed* attribute is a flag to determine whether or not the order was carried out. (If two orders conflict, they will not be carried out, but they will be preserved in the game history.) Orders are related to countries via the *Country-Gives-Order* relationship because all actions in the game are carried out via countries, and we may display order history.

Orders occasionally have arbitrary numbers of arguments, i.e. Move-Via-Convoy. This case is handled by the *Operand* entity.

*Operand(opr\_id:int, ord\_id:int, ter\_id:int)*

The *Operand* entity allows the program to supply multiple arguments to an order. Operands contain their own ID, the ID of their owner, and the territory argument. The order of the arguments does not matter, so we do not need to insert them in a consistent order. The *ord\_id* foreign key implements the *Operand-RefersTo-Order* relation, and the *ter\_id* foreign key implements the *Territory-Participates-Operand* relation.

*User(usr\_id:string, name:string, email:string, screen\_name:string, pass\_hash:string, salt:string, last\_login:datetime, creation:datetime, status:string)*

When a person creates an account, they enter their own name, email address, screen name, and password. The system sets *usr\_id* by generating a *32*-byte SHA-2 hash based on random seed data provided by /dev/urandom. The password salt is generated the same way from different seed data. The account creation date is stored in *creation*, and *last\_login­* is updated whenever they log into the system. The *status* attribute holds a status message to display to the user at the bottom of the screen.

*Session(session\_id:string, sig\_id:string, msg\_sig:string, usr\_id:string, last\_update:datetime)*

Whenever a user logs in, a *Session* is created. The *session\_id* attribute is a SHA-2 string. The *sig\_id* and *msg\_sig* attributes are session spoofing prevention measures that ensure the validity of the user’s session. The *last\_update* attribute is updated whenever the user loads a new page in the system.

*GameMembership(usr\_id:string, gam\_id:int, orders\_given:int)*

Users can play more than one game at once, and their past games are saved in the database. This setup constitutes a many-to-many relationship which requires a separate entity, *GameMembership*. This entity stores foreign keys to *User* and *Game*. In addition to representing the relationship, the *GameMembership* entity keeps track of whether players have clicked the “Issue Orders” button indicating that they are finished choosing orders. If the *orders\_given* attribute is set for all players and the game click has not run out, then the turn will roll over anyway in order to avoid wasting time.

*Message(msg\_id:int, from\_usr:string, to\_usr:string, time\_sent:datetime, subject:string, msg:string, read:int)*

Since discussion is an integral part of Diplomacy, and therefore Diplomacloud, the ability to communicate is essential. The *Message* entity is the main entity in the messaging system. Each message has an ID, attributes for the sender and recipient, the time sent, a subject line, a message body, and a flag to determine if the message has been read by the recipient. We do not plan to implement messages with multiple recipients; instead, we will create copies for each individual.

**Queries**

Generating the map:

•Create new instances of *Line, Triangle, Territory, Adjacent, LineInTerritory,* and *Map*

based on the data generated by the map generator program

Drawing the map:

•Select all lines associated with the chosen map

•Select all triangles associated with the map

•Select all territories associated with the map

•Join territories with countries to obtain their colors

•Update the *Game.pic* attribute to point to the current location of the map

These data will be used to draw and color the triangles and then inform the database of

where the new image is saved.

Starting the game:

•Get a list of all users

•Get a list of users that are members of the current game

•Get the *pic* attribute of the chosen map

•Update the *Game.map\_id* attribute

•Add GameMembership entries

•Create new instances of *Country*, *GamePiece*, and *Supplier*

•Update the *Game.turn\_start, turn\_length, turn\_stage, season,* and *year* attributes

Turn stage 1:

•Select all possible Stage 1 orders from the order\_types table

•Select all sea territories associated with the current map

•Select all territories adjacent to a specified territory

•Create new instances of *Order* and *Operands*

Turn stage 2:

•Select all orders in this turn

•Select all orders in this turn with the same destination

•Update *Order.executed* attributes for orders that did not conflict

•Update instances of *GamePiece* referenced by orders with *executed* attributes set

•Select all supply centers that belong to each player one at a time

Turn stage 3:

•Select all territories adjacent to a specified territory

•Select a specified *GamePiece*

Turn stage 4:

•Select all orders in this turn

•Update instances of *GamePiece* referenced by orders

Turn stage 5:

•Select all supply centers belonging to a country

•Remove and create instances of *GamePiece* based on orders

End game/old game view:

•Select *Game* to display statistics

•Select all supply centers belonging to a country

User activities:

•Create, update, and delete instances of *Session*

•Log out of a *Session*

•Create a new instance of *User*

•Update the *last\_login* attribute of a *User* instance

•Select all messages with a specific *User* as the recipient

Message activities:

•Create and delete instances of *Message*









