CHAPTER 3

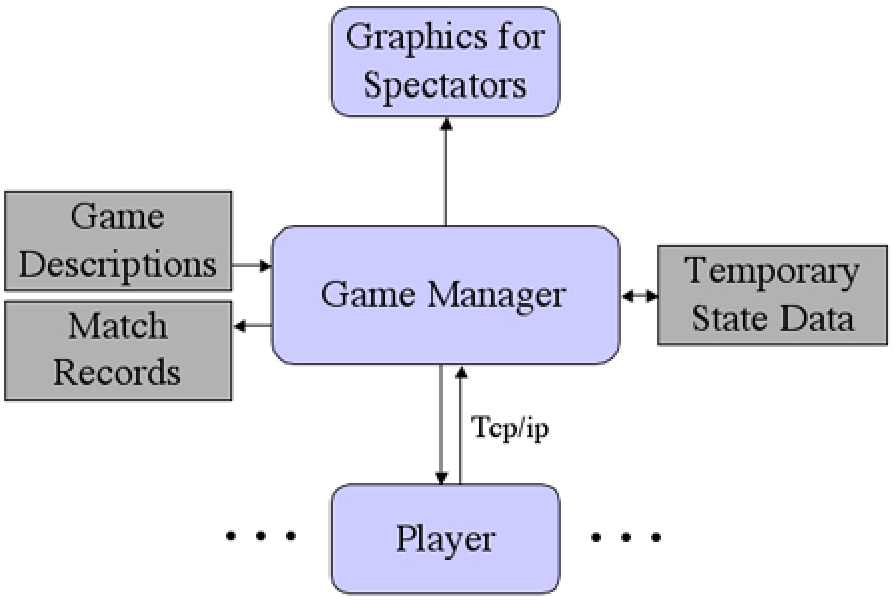
Game Management

3.1 Introduction

This chapter is an overview of game management. More properly, it should be called match management, as the issue is how to manage individual matches of games, not the games themselves. We start with an overview of the General Game Playing ecosystem and the central role of the Game Manager. We then discuss the General Game Playing communication protocol. Finally, we see how it is used in a sample game.

3.2 Game Management

A diagram of a typical general game playing ecosystem is shown below. At the center of the ecosystem is the game manager. The game manager maintains a database of game descriptions and match records, and it maintains some temporary state for matches while they are running. The game manager communicates with game players. It also provides a user interface for users who want to schedule matches, and it provides graphics for spectators watching matches in progress.



3.3 Game Communication Language

Communication between the Game Manager and game players takes place through HTTP connections. The communication model assumes that each player is running on an Internet-connected host listening on a particular port. HTTP messages sent to players have the standard HTTP header, with content type `text/acl`.

In the current GGP communication language, there are five types of messages used for communication between the Game Manager and game players.

(1) An `info` message is used to confirm that a player is up and running. The simplest form is shown below.

```
info()
```

Upon receipt of an `info` message, a player is expected to return `available` if it is ready to receive messages. Otherwise, it should return `busy`. It may also return arbitrary information about itself as a sequence of pairs of type and value, e.g. `[[name,egghead],[status,available]]`.

(2) A `start` message is used to initialize an instance of a game. The general form of a `start` message is shown below. The message begins with the keyword `start`, and this is followed by five arguments, viz. a match identifier (a sequence of alphanumeric characters beginning with a lower case letter), a role for the player to play (chosen from the roles mentioned in the game description), a list of game rules (written as sentences in GDL), a start clock (in seconds), and a play clock (also in seconds).

`start(`*id*`,`*role*`,`*description*`,`*startclock*`,`*playclock*`)`

Upon receipt of a `start` message, a player should prepare itself to play the match. Once it is done, it should reply `ready` to tell the Game Manager that it is ready to begin the match. The GGP protocol requires that the player reply before `startclock` seconds have elapsed. If the Game Manager has not received a reply by this time, it will proceed on the assumption that the player is ready.

(3) A `play` message is used to request a move from a player. The general form of the `play` message is shown below. It includes an identifier for the match and a record of the moves of all players on the preceding step. The order of the moves in the record is the same as the order of roles in the game description. On the first request, where there is no preceding move, the `move` field is `nil`.

`play(`*id*`,`*move*`)`

Upon receipt of a `play` message, a player uses the move information to update its state as necessary. It then computes its next move and returns that as answer. The GGP protocol requires that the player reply before `playclock` seconds have elapsed. If the Game Manager has not received a reply by this time *or* if it receives and illegal action, it substitutes a random legal action.

(4) A `stop` message is used to tell a player that a match has reached completion. The general form of the `stop` message is shown below.

`stop(`*id*`,`*move*`)`

Upon receipt of a `stop` message, a player can clean up after playing the match. The `move` is sent along in case the player wants to know the action that terminated the game. After finishing up, the player should return `done`.

(5) An `abort` message is used to tell a player that a match is terminating abnormally. It differs from a `stop` message in that the match need not be in a terminal state.

`abort(`*id*`)`

Upon receipt of an `abort` message, a player can eliminate any data structures and return to a ready state. Once it is finished, it should return `done`.

3.4 Game Play

The process of running a match goes as follows. Upon receiving a request to run a match, the Game Manager first sends a `start` message to each player to initiate the match. Once game play begins, the manager sends `play` messages to each player to get their plays; and it then simulates the results. This part of the process repeats until the game is over. The Manager then sends a `stop` message to each player.

Here is a sample of messages for a quick game of Tic Tac Toe. The game manager initiates the match by sending a `start` message to all of the players, each with a different role. The players then respond with `ready`. They can respond immediately or they can wait until the start clock is exhausted before responding.

> Game Manager to White: `start(m23,white,[role(white),role(black),...],10,10)`
> Game Manager to Black: `start(m23,black,[role(white),role(black),...],10,10)`
> White to Game Manager: `ready`
> Black to Game Manager: `ready`

Play begins after all of the players have responded or after the start clock has expired, whichever comes first. The manager initiates play by sending a `play` message to all of the players. Since this is the first move and there are no previous moves, the move argument in the play message is `nil`. In this case, the first player responds with the action `mark(1,1)`, one of its nine legal actions; and the second player responds with `noop`, its only legal action.

> Game Manager to White: `play(m23,nil)`
> Game Manager to Black: `play(m23,nil)`
> White to Game Manager: `mark(1,1)`
> Black to Game Manager: `noop`

The Game manager checks that these actions are legal, simulates their effects, updates the state of the game, and then sends `play` messages to the players to solicit their next actions. The second argument in the `play` message this time is a list of the actions received in response to the preceding `play` message. On this step, the first player responds with `noop`, its only legal action; and the second player responds with `mark(1,2)`. This is a legal move, but it is not a wise move, as the game is now strongly winnable by first player.

> Game Manager to White: `play(m23,[mark(1,1),noop])`
> Game Manager to Black: `play(m23,[mark(1,1),noop])`
> White to Game Manager: `noop`
> Black to Game Manager: `mark(1,2)`

Again, the Game Manager checks legality, simulates the move, updates its state, and sends a `play` message requesting the players' next actions. The first player takes advantage of the situation and plays `mark(2,2)` while the second player does `noop`.

> Game Manager to White: `play(m23,[noop,mark(1,2)])`
> Game Manager to Black: `play(m23,[noop,mark(1,2)])`
> White to Game Manager: `mark(2,2)`
> Black to Game Manager: `noop`

There is not much the second player can do in this situation to save itself. Instead of staving off the immediate loss, it plays `mark(1,3)`, while the first player does `noop`.

> Game Manager to White: `play(m23,[mark(2,2),noop])`
> Game Manager to Black: `play(m23,[mark(2,2),noop])`
> White to Game Manager: `noop`
> Black to Game Manager: `mark(1,3)`

The Game manager again simulates, updates, and requests a move. In this case, the first player goes in for the kill, playing `mark(3,3)`.

> Game Manager to White: `play(m23,[noop,mark(1,3)])`
> Game Manager to Black: `play(m23,[noop,mark(1,3)])`

White to Game Manager: `mark(3,3)`
Black to Game Manager: `noop`

With this move, the game is over. As usual, in such cases, the Manager lets the players know by sending a suitable `stop` message. It then stores the results in its database for future reference and terminates.

Game Manager to White: `stop(m23,[mark(3,3),noop])`
Game Manager to Black: `stop(m23,[mark(3,3),noop])`
White to Game Manager: `done`
Black to Game Manager: `done`

Note that the Game Manager sends slightly different `start` messages to the different players. Everything is the same except for the role that each player is asked to play. In all other cases, the same messages are sent to all players. In advanced versions of the General Game Playing protocol, this symmetry is broken. The Game Manager can send different game descriptions to different players. And it can tell different players different information in the `play` and `stop` messages.

Exercises

Exercise 3.1: Consider the game of Tic Tac Toe given in the preceding chapter. Assume that a Game Manager has sent `start` messages to the players of a match with name `m23` and with the rules from the preceding chapter as game description; and assume that the players have just replied that they are ready to play. Which of the following is the correct message for the Manager to send to the players next?

(*a*) `play()`
(*b*) `play(m23)`
(*c*) `play(m23,nil)`
(*d*) `play(m23,noop)`
(*e*) `play(m23,[mark(1,1),noop])`

Exercise 3.2: Consider the game of Tic Tac Toe given in the preceding chapter. Assume that the game is in the state shown on the left below, and assume that the manager has just received the action `(mark 2 2)` from the first player and the action `noop` from the second player. Which of the messages shown on the right is the correct message to send to the first player?

X	O	

(*a*) `play()`
(*b*) `play(m23)`
(*c*) `play(m23,mark(2,2))`
(*d*) `play(m23,noop)`
(*e*) `play(m23,[mark(2,2),noop])`

Exercise 3.3: Consider the game of Tic Tac Toe given in the preceding chapter. Assume that the game is in the state shown on the left below, and assume that the manager has just received the action `mark(2,2)` from the first player and the action `noop` from the second player. Which of the messages shown on the right is a correct message to send to the first player.

X	O	
	X	O

(*a*) `play()`
(*b*) `play(m23)`
(*c*) `play(m23,mark(2,2))`
(*d*) `play(m23,noop)`

(*e*) `play(m23,[mark(2,2),noop])`

(*f*) `stop(m23,[mark(3,3),noop])`

(*g*) `stop(m23,[mark(2,2),noop])`

(*h*) `abort(m23)`