[DRAFT] Kalah Game Protocol

Kaludercic, Philip   Völk, Tobias

Abstract

This document specifies a protocol for playing the game Kalah, a member of the Mancala family. It has been designed to be modularized, so that not all implementations have to implement all features. The main modules presented here are freeplay, evaluation and validation.

This document specified version 1.1.0 of the KGP protocol.

Contents

1 Prelude

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “NOT RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.1 Definitions

A server organizes activities between one or more clients. The server waits for clients to request an activity, that the server may or may not organize. Activities cannot be changed, after they have been requested.

The server and the client communicate using a text-based, line-oriented protocol, over a reliable, ordered and error-checked transport layer (e.g. TCP).

1.2 Formal Structure

The protocol consists of commands sent between client and server. Server-to-client and client-to-server commands have the same form, consisting of:

- Optional, unique command ID. Client and server MUST ensure that no ID is reused.
- Optional command reference (addressing a previous command ID). The client MAY NOT reference a non-existing command ID.
- A command name
- A number of arguments

The ABNF representation of a command is as follows:

```
command  = id name *(*1WSP argument) CRLF
id       = [[*1DIGIT] [ref] *1WSP]
ref      = ["@" *1DIGIT]
```

```
name     = *1(DIGIT / ALPHA)
argument = integer / real / word
         / string / board
integer  = [("+" / "-")] *1DIGIT
real     = [("+" / "-")] *DIGIT "." *1DIGIT
word     = *1(DIGIT / ALPHA / "-" / ":")
string   = DQUOTE scontent DQUOTE
scontent = *("\" CHAR / NDQCHAR)
board    = "<" *1DIGIT *("," *1DIGIT) ">"
```

where `NDQCHAR` is every `CHAR` except for double quotes, backslashes and line breaks. Each command MUST at most be most 16384 characters long, including trailing white space. Any line beyond that MAY be ignored by a server.

An argument has a statically identifiable type, and is either an integer (`32`, `+0`, `-100`, ...), a real-valued number (`0.0`, `+3.141`, `-.123`, ...), a string (`single-word`, `"with double quotes"`, `"like \" this"`, ...) or a board literal.

Board literals are wrapped in angled-brackets and consist of a an array of positive, unsigned integers separated using commas. The first number indicates the board size n, the second and third give the number of stones in the south and north Kalah respectively. Values 4 to $4+n$ list the number of stones in the south pits, $4+n+1$ to $4+2n+1$ the number of stones in the north pits:

```
<3,10,2,1,2,3,4,2,0>
 ^ ^  ^ ^     ^
 | |  | |     |
 | |  | |     \__ North pits: 4, 2 and 0
 | |  | \________ South pits: 2, 1 and 3
 | |  \__________ North Kalah
 | \_____________ South Kalah
 \_______________ Board Size
```

1.3 Protocol Overview

The communication MUST begin by the server sending the client a `kgp` command, with three arguments indicating the major, minor and patch version of the implemented protocol, e.g.:

```
kgp 1 0 1
```

The client MUST parse this command and that it implements everything that is necessary to communicate. The major version indicates backwards incompatible changes, the minor version indicates forwards incompatible changes and the patch version indicates minor changes. A client MAY only check the major version to ensure compatibility, and MUST check the minor and patch version to ensure availability of later improvements to the protocol.

The client MUST eventually proceed to respond with a `mode` command, indicating the activity it is interested in. The `mode` command is REQUIRED to have one string-argument, indicating the activity.

```
mode freeplay
```

In case the server doesn't recognize or support the requested activity, it MUST immediately indicate an error and close the connection:

```
error "Unsupported activity"
goodbye
```

The detail of how the protocol continues depends on the chosen activity. The server SHOULD terminate the connection with a `goodbye` command.

After the connection has been established and version compatibility has been ensured, the server MAY send a `ping` command. The client MUST answer with `pong`, and SHOULD do so as quickly as possible. In absence of a response, the server SHOULD terminate the connection.

Both client and server MAY send `set` commands give the other party hints. Both client and server SHOULD try to handle these, but MUST NOT terminate the connection because of an unknown option. Version commands indicating capabilities and requests SHOULD be handled between the version compatibility is ensured (`kgp`) and the activity request (`mode`).

Any command (client or server) MAY be referenced by a response command: `ok` for confirmations and `error` for to indicate an illegal state or data. All three MUST give a semantically-opaque string argument. The interpretation of a response depends on the mode.

2 Defaut Modes

The following sections shall specify modes ("activities") that a client SHOULD be able to request from any server. Further modes MAY be supported, but they are not specified here.

2.1 Freeplay Mode

The "freeplay" involves the server sending the client a sequence of board states (`state`) that the client can respond to (`move`). The server MAY restrict the time a client has to respond (`stop`), that the client MAY also give up by their own accord (`yield`). IDs and references SHOULD be used to ensure the correct and unambitious association between requests and answers.

A server might use the `freeplay` mode to implement a tournament, as seen in this example:

```
s: kgp 1 0 1
c: mode freeplay
s: 4 state <3,0,0,3,3,3,3,3,3>
c: @4 move 1
s: 6@4 stop
s: 8 state <3,1,3,0,4,4,4,3,3>
c: @8 move 3
c: @8 move 2
c: @8 yield
s: 10@8 stop
...
```

Where `s:` are commands sent out by the server, and `c:` by the client.

There are no requirements on how a server is to send out `state`-requests and on how long the client is given to respond.

3 Freeplay commands

The following commands must be understood for a client to implement the "freeplay" mode:

`state [board]` (server) Sends the client a board state to work on. The command SHOULD have an ID so that later `move`, `yield` and `stop` commands can safely reference the request they are responding to, without interfering with other concurrent requests.

The client always interprets the request as making the move as the "south" player.

`move [integer]` (client) In response to a `state` command, the client informs the server of their preliminary decision. Multiple `move` commands can be sent out, iteratively improving over the previous decision.

An integer n designates the n'th pit, that is to say uses 1-based numbering. The value must be in between $1 \leq n < s$, where s is the board size.

`stop` (server) An indication by the server that it is not interested in any more `move` commands for some `state` request. Any `move` command sent out after a `stop` MUST be silently ignored.

If the client has not sent a `move` command, the server MUST make a random decision for the client.

`yield` (client) The voluntary indication by a client that the last `move` command was the best it could decide, and that it will not be responding to the referenced `state` command any more. The client sending a `yield` command is analogous to a server sending `stop`.

3.1 Evaluation Mode

The "evaluation" mode involves the client giving numerical evaluations for given states. An evaluation is a real-valued number, without any specified meaning. The client SHOULD be consistent in evaluating states (the same board should be approximately equal, a board with a better chance of winning should have a better score, ...).

After requesting the mode with

```
mode eval
```

the server may immediately start by sending `state` commands as specified for the "freeplay" mode.

3.2 Evaluation commands

`state [board]` (server) See "Freeplay commands". The server MUST send a command ID.

`eval [real]` (client) The client MUST reference the ID of the `state` command it is evaluating. Multiple commands can be sent out in reference to one `state` request.

`stop` (client) See "Freeplay commands". The server MUST use a command reference. The client SHOULD stop responding to the referenced `state` request.

3.3 Verification Mode

To ensure that clients don't misinterpret the rules of Kalah, they can request this game mode and have the server challenge them with random game states that they should compute.

A client does this by initially sending

```
mode verify
```

3.4 Verification commands

`problem [board] [move]` (server) The server send the valid board state and a legal move. The client will respond to this using `solution`. The server MUST send a command ID.

`solution [board] [integer]` (client) A response to a `problem`. The client sends back the resulting board state and an indication whether or not the move was a repeat move or not (0 for false and non-0 for true). The client SHOULD use a command ID.

The server will respond to this message with an `erorr` message in case the client made a mistake.

4 Responses

5 The `set` Command

The `set` command may be used at any time by both client and server to inform the other side about capabilities, internal states, rules, etc. The structure of a set command is

```
set [option] [value]
```

Each option is structured using colons (`:`) to group commands together. Each command group specified here SHOULD be implemented entirely by both client and server:

5.1 `info`-group

On connecting, server and client may inform each other about each other. The options of this group are:

`info:name` (string) The codename of the client or the server.

`info:authors` (string) Authors who wrote the client

`info:description` (string) A brief description of the client's algorithm.

`info:comment` (string) Comment of the client about the current game state and it's chosen move. Might contain (depending on the algorithm), number of nodes, search depth, evaluation, . . .

5.2 `time`-group

For "freeplay" and especially "simple", the server may indicate how it manages the time a client is given. The options of this group are:

`time:mode` (word) One of `none` when no time is tracked, `absolute` if the client is given an absolute amount of time it may use and `relative` if the time used by a client for one `state` request has no effect on the time that may be used for other requests.

`time:clock` (integer) Number of seconds a client has left. This option MAY be set by the server before issuing a `state` command.

`time:opclock` (integer) Number of seconds an opponent has left.

5.3 `auth`-group

In cases where an identity has to be preserved over multiple connections (a tournament or other competitions), some kind of authentication is required. The `auth` group consists of a single variable to implement this as simply as possible:

`auth:token` (string) As soon as the client sends sets this option, the server will associate the current client with any previous client that has used the same token. No registration is necessary, and the server MAY decide to abort the connection if the token is not secure enough.

The value of the token must be a non-empty string.

`auth:forget` (string) Request that the server forgets a client associated with a token. The token MAY NOT be known to the server, and the server SHOULD NOT directly indicate if the request succeeded.

The client SHOULD use an encrypted connection when using the auth group, as to avoid MITM attacks. The server MUST NOT reject connections that do not set `auth:token`.

5.4 `game`-group

As neither "freeplay" nor "simple" mode guarantee a logical sequence of `state` commands, that might represent a possible game, the agent cannot assume that two consecutive state commands represent the chronological development of a game between the south and north sides.

In case the server internally matches two clients against one another, and sends these a logical sequence of `state` commands, the `game` group may be used to indicate this.

The options this groups offers are:

`game:id` (string) An opaque identifier to represent a logical game. The option MUST be set before a `state` command has been sent. The client MAY then associate `state` commands with the same `game:id` annotations and assume them to be a sequence of game states.

Two consecutive state commands with the same `game:id` MUST represent two game states. An empty string indicates an anonymous game.

`game:uri` (string) A URI pointing to a resource that describes the current game in more detail. The resource should be publicly accessible, or provide the necessary credentials for the client to access it.

An empty string indicates there is no URI for this game.

`game:opponent` (string) A name of the opponent the client is playing against. The name SHOULD be unique. Interpreted the same way `game:id` is. An empty string indicates an unknown opponent.

6 Notes

This section is non-normative.

The intention of the KGP protocol is to provide a simple, extensible yet forward compatible to implement language for AI applications.

7 Distribution of This Document