On the Designing of a Text Protocol for the Game of Kalah

Philip Kaludercic
philip.kaludercic@fau.de
https://wwwcip.cs.fau.de/~oj14ozun/

September 15, 2022

Abstract

With computers dominating most abstract board games, the use of a protocols has become common. This makes comparisons between various implementations possible, without depending on specific technologies or programming languages.

This paper reports on the process of *developing a text protocol* for the game of "Kalah", a member of the "Mancala" together with a technical analysis of a tournament that took place in the context of the AI1 course (Artificial Intelligence course focusing on symbolic methods) at the University of Erlangen–Nuremberg.

Contents

1 Motivations and Values for Game Protocols 1
- 1.1 Survey of existing protocols 2
  - 1.1.1 Chess 2
  - 1.1.2 Go (Baduk) 2
  - 1.1.3 General Game Playing 3
- 1.2 General Considerations 4
  - 1.2.1 The Protocol Syntax 4
  - 1.2.2 Connection Persistence . . . 4
  - 1.2.3 Extensibility 5

2 Design of the KGP Protocol 5
- 2.1 Background and Context 5
- 2.2 The Protocol 6
  - 2.2.1 Version Agreement 6
  - 2.2.2 Protocol Modes 6
  - 2.2.3 Command IDs and References 6
  - 2.2.4 The `set`-Command 7
- 2.3 Rationale 8

3 Server and Client implementations 8
- 3.1 Clients 8
  - 3.1.1 Python Client 8
  - 3.1.2 Java Client 9
  - 3.1.3 Wrapper client 9
- 3.2 Server 9

4 Technical Evaluation of a Tournament 10
- 4.1 Transport Layer 10
- 4.2 Authentication 10
- 4.3 The `simple`-mode 11
- 4.4 Evaluation of the Closed Tournament 11

5 Future Thoughts and Plans 11

A The "Kalah Game Protocol" Specification 13

1 Motivations and Values for Game Protocols

In order to provide the greatest freedom while comparing the performance of various programs, it is useful to provide a language agnostic protocol as a common denominator. The role of the protocol is to formally specify what input to expect and what output to produce. This approach isolates the details of a concrete implementation down to a black box that only exposes the necessary functionality.

This freedom is of particular interest in competitive situations, where participants should not be burdened by unnecessary restrictions such as programming languages or runtime environments, that might limit or hinder possible implementation strategies.

In this section we shall begin by considering a few existing protocols for competitive, abstract games

to see what decisions are made, then follow by a brief abstract analysis of the matter.

1.1 Survey of existing protocols

1.1.1 Chess

Since the triumph of IBM's *Deep Blue* AI against Garry Kasparov in 1997, it has been evident that in the coming future AI would play a dominant role in the world of chess. It is at least at this point that computer-to-computer communication becomes necessary, as the best "players" are all virtual.

In this space there have been two significant attempts: One of the first such examples was the Chess Engine Communication Protocol[13] that began being developed around the mid 1990s by the maintainers of the GNU XBoard graphical interface for playing chess. It was initially designed as a frontend for GNU Chess (the chess engine implementation by the GNU Project), but as lead maintainer, Tim Mann explained in an interview[10], the requests of other chess engine developers to be supported by XBoard too, resulted in an "extending the ad-hoc engine protocol to support them". The consequence of this approach was that in a matter of only a few years it became obvious the protocol had to change:

> Unfortunately, because the protocol was never really designed, but just grew out of documenting the existing communication with GNU Chess, there are still several bugs and deficiencies in it today. It would be nice to make some major revisions, but then of course it would (at best) take a long time for the existing engines to convert over to the new protocol, so both would have to be supported, probably forever.[10]

This demonstrates the danger of relying too much the organic growth of a protocol over time. It appears some conscious, *a priori* planning is necessary to avoid these issues like these.

The eventual successor to Chess Engine Communication Protocol was the Universal Chess Interface (UCI)[7]. While influenced by the former, the primary technical advantages over the previous protocol were changes that improved robustness. One way this was done was by trying to make the protocol "stateless". This means that messages avoid relying on previous communication for interpretation. Another significant change was a general attempt to simplify the "deficiencies" that the Chess Engine Communication Protocol had suffered from.

UCI, like the Chess Engine Communication Protocol before it, is a plain-text protocol. That is to say that each time messages are exchanged, they are encoded the same way a human would write text in a plain text file. Take for example this command, that a user agent (GUI, server, ...) might send to a chess engine to the initial position of the board:

```
position startpos moves e2e4
```

The chess engine receives this command, parses and interprets what is to be done, and then proceeds to await the next command, such as

```
go movetime 1000
```

to request a move be calculated given milliseconds 1000 of time to consider. It then responds with a message such as[1]

```
bestmove d7d5 ponder e4d5
```

Nowadays, UCI is the *de facto* standard protocol used by chess engines.

1.1.2 Go (Baduk)

As with Chess, the east-Asian game of Go (otherwise also known as "Igo", "Baduk" or "Weiqi") has received a lot of attention. It has long been regarded as a worthwhile vehicle to study adversarial searching and artificial intelligence. All the more so, after the *Deep Blue*–Garry Kasparov matches, the reputation of being a stronghold of human intellectual superiority over the machine was shifted from Chess to Go.

This was the case until "AlphaGo"[12], a Go program by DeepMind, managed to beat Lee Sedol in 2016. At the time he was considered the highest ranking, professional Go player in the world. It was at least at this point that having a standardised means of communicating between programmes was necessary.

[1] This response was generated using GNU Chess, Version 6.2.9.

As with Chess before, there are two protocols worthy of notice. The first is the Go Modem Protocol (GMP)[15]. The details on the exact history are sparse, and it appears to be displaced by the simpler Go Text Protocol (GTP)[3], initially introduced by the GNU Go Project (the Go engine implementation by the GNU Project)[1, Section 19].

On a superficial level, these two differ in that GMP is a "binary" protocol, where singular bits are used to communicate intent, while GTP is once again a "plain-text" protocol. As an example, if the user agent wishes to have the white player place a stone on the field B1 (the uppermost, second from the left intersection), GMP specifies that the command be formatted using only two bytes, here the binary representation:

```
1101 0100  1000 0010
```

while GTP equivalent is written out on a single line terminated with carriage return and new line characters:

```
play white 1B
```

GTP is obviously more legible and makes debugging easier, yet GMP is not without its advantages. The immediate one is reduced traffic, as the "language" is inherently compressed down to only what is necessary. Another advantage that is not directly visible from this example is that despite the terse communication, provisions have been made for the possible need to extend the protocol in the future, while not breaking compatibility with older implementations (this shall be referred to later as "weak extensibility").

Nevertheless, GTP appears to have established itself as the more popular choice between the two, which could speak for the inherent advantage of a plain text protocol over a binary alternative — especially when the performance advantages make little to no difference[2].

A final feature of GTP worth noting is the command-response-error structure. This allows for a command to be prefixed with an ID, and later commands to reference it and either indicate some success or failure. On this basis asynchronous communication is made easier, as it isn't necessary to await a response before sending out a new message.

[2] An example where the advantage was significant enough is HTTP. For the first approx. 20 years of the WWW, sending headers as plain text was sufficient, but with the ever-increasing traffic, HTTP 2.0 and newer switched over to binary encoding to reduce the amount of transferred data and accelerate parsing. In modern networks, neither of these issues present themselves as bottlenecks in game protocols like GTP.

1.1.3 General Game Playing

A slightly different domain from the previous two examples is that of General Game Playing (GGP). This problem involves the attempt to design an algorithm that can play any game without any prior knowledge, as long as a declarative, formal specification can be given [5, p. 1].

One such specification format is the Game Description Language (GDL), developed by the Stanford Logic Group for "finite, discrete, deterministic multi-player games of complete information"[8, p. 1]. The specification is written in Datalog (a subset of Prolog, c.f. [9]) formatted using Knowledge Interchange Format (KIF). KIF in turn is a formal language for the "interchange of knowledge among desperate computer programms"[6, p. 5], that uses S-Expression syntax [5, 7 ff.] (as defined by [11, 186 ff.]). For example, the following fragment from [8, p. 18] describe the legal moves a player may make (symbols beginning with a leading ? are variables, those without are constants, analogous to upper and lower case symbols in Prolog respectively):

```
(<= (legal ?player (mark ?x ?y))
    (true (cell ?x ?y b))
    (true (control ?player)))

(<= (legal x noop)
    (true (control o)))

(<= (legal o noop)
    (true (control x)))
```

GDL was explicitly designed for competitions, as seen by the intended infrastructure that GDL is employed by. [5, 69 ff.] describes a centralized structure where a "game manager" (server) takes the role of an arbiter between multiple "players" (clients). These communicate via HTTP requests, where a client first requests to play a game and receives a GDL specification and a new ID for a match. This ID is necessary for the game manager to identify and associate a player with previous requests.

The protocol is notably asymmetric, as requires a centralized network architecture that doesn't provision clients to connect directly to one another. Furthermore, connections can only be initiated *by* a "player" *towards* a "game manager", not the other way around.

1.2 General Considerations

The following section will consider both advantages and disadvantages of various choices a protocol designer makes, in an abstract manner.

1.2.1 The Protocol Syntax

Except for GMP and GDL, each protocol is based on plain-text, bi-directional line-oriented communication. That is to say that a command is written entirely on a single line, and is usually terminated by either a carriage return and a newline (`\r\n` in C notation) or just a newline (`\n`). This is a popular approach, not only employed by game protocols, but by many internet standards such as SMTP, FTP, IRC or HTTP (up until version 2.0).

As previously discussed, GMP demonstrates an alternative, namely to interpret messages on a byte-by-byte basis. This reduces the necessary amount of traffic, accelerates parsing, but makes it more difficult to follow along. It appears as though the disadvantages of this approach outweigh the advantages, as mentioned in footnote 2.

Finally, we see that GDL re-uses the existing syntax of KIF, which in turn is intentionally meant to be legal Common Lisp syntax. The GDL specification explains[8, p. 7]:

> In particular, a string of ascii characters forms a legal expression in [...] KIF if and only if (1) if it is acceptable to the Common Lisp reader [...] and (2) the structure produced by the Common Lisp reader is a legal expression of structured KIF

This is particularly convenient for a Common Lisp programmer, who can just re-use the read `(read)`[3] procedure to parse the input, but acceptable in other languages as well that can re-use an existing parser, or if not available have to implement it themselves. The same applies to other data interchange formats such as JSON, XML or MessagePack. As with S-Expressions, one can also claim that each format has a certain bias towards the cultures that they emerged from. How good of a choice this approach is depends on the context and format one is confronted with.

[3] `http://www.lispworks.com/documentation/HyperSpec/Body/f_rd_rd.htm`

1.2.2 Connection Persistence

GDL once again distinguishes itself from the other protocols by not requiring a persistent connection, whilst simultaneously specifying that HTTP must be used to share messages.

The Chess Engine Communication Protocol, UCI and GTP only require a *reliable* transport layer such as TCP[2], UNIX domain socket or regular IPC mechanisms (such as Pipes) to exchange messages.

GMP, as the name implies ("Modem"), makes explicit, low-level assumptions about the manner clients communicate and what serial ports which player connects to, see [1, Subsection 3.7]. It is reasonable to assume that requirements like these contributed to the loss of popularity, compared to GTP.

The principal considerations therefore are whether the abstract *laissez-faire* approach of UCI and GTP is preferable to the fixed decisions that GDL has already made, and what the value of connection persistence is.

One issue that must be considered in practice, is that when communicating over networks, the first of L. Peter Deutsch's *Fallacies of distributed computing*, stating "The network is reliable". Any long-standing connection, even if it relies on TCP or similar technologies, is prone to sudden and often not immediately noticed outages. The more complex the network topology, the higher the risk is, due to more points of possible failure.

On the other hand, if the network is already experiencing issues, the necessity to re-establish new connection for every message, bundled with the necessary overhead that entails, does not improve the situation. For both approaches to therefore be equal in terms of functionality and capabilities, it is necessary for persistent connections to identify themselves, as it the case by necessity with non-persistent protocols.

Otherwise, the inherent freedom to use any reliable connection appears preferable, as this allows for more flexibility to adapt to unforeseen circumstances and requirements.

1.2.3 Extensibility

Along the same lines, the flexibility of a protocol to adapt to changing requirements is desirable. Lacking this or the capabilities to be updated in a forward-compatible way, is what lead to the demise of Chess Engine Communication Protocol and the rise of UCI.

Though it should be noted that Extensibility is a double-edged sword. An overemphasis can quickly become more of a burden by means of fragmentation than a virtue.

To take an example from a different domain, Extensible Messaging and Presence Protocol (XMPP, formerly also known as "Jabber") is a federated, instant messaging protocol that enjoyed widespread use and popularity in the midaughts. The protocol is extended through a formal process, that involves publishing so called "XEP" (XMPP Extension Protocols) reports. With the rise of mobile computing these became more and more important, as the assumptions the early designers had operated under in the late 1990s were not given anymore. These changes included that each user only had one device, that devices might quickly appear and disappear from the network and that a persistent connection might require more power than is acceptable. The attempt was made for to address these issue and at the same time up update the capabilities of the protocol to satisfy the new assumptions users had about what an instant messaging service should provide (multi-user chat rooms, location sharing, avatars, file uploading, encryption,...).

Compared to centralized and commercial alternatives, this process was notably slow, which made the network less appealing to day-to-day users. Moreover, the fact that multiple XEPs were being proposed to solve the same issues lead to fragmentation and increased the complexity of implementing a client or a server. The situation has improved over the last few years, but remains as an important reminder that careless extensibility can harbor. Nevertheless, one should keep in mind that if it were not for the flexibility inherent in the protocol, none of these improvements would have been possible in the first place.

2 Design of the KGP Protocol

This section will present and attempt to legitimate the decisions made in designing a protocol for the abstract board game of "Kalah". We shall begin with a presentation of the context and its related requirements. Given this background we will proceed to a brief overview of the protocol itself and finally conclude with reflections on the decisions made in reference to the observations made in section 1.2.

2.1 Background and Context

The interest in a protocol for Kalah stems from the "Artificial Intelligence 1" course[4], held at the Friedrich-Alexander University in Erlangen. The course is accompanied by an exercise slot, where students are given an opportunity to apply their knowledge in practice. In one particular exercise, the task is given to implement an agent that is able to play the abstract board game "Kalah". Students can then decide to have their implementations take participate in a tournament. We will go into the details of this tournament in section 4.

For the current intent, the rules and detail of "Kalah" are not relevant. The only important facts are that it is an abstract, discrete, deterministic, two-player game, where each player makes a move that can be described by a bounded number. Further details on the game are elaborated in [14].

The proposal to use a protocol was made as a possible replacement for the previous framework that was used in the exercise[5]. The previous framework was written in Scala, and implementing an agent involved inheriting and implementing an abstract base class. This restricted clients to being implemented in JVM-based languages (Scala, Java,...) and reduced the flexibility they had in terms of initialization. The concrete implementation of the framework also suffered issues, and could in retrospect be argued to have been ill-designed. This became apparent when the Framework was extended to allow for the agent to "update" their decision, that is to say if the agent was given 5 seconds to make a move, they would be able to improve their decision until the time was up. The issue was that this was implemented using a shared variable, that

[4] `https://kwarc.info/courses/ai1/`

[5] c.f. `https://github.com/KWARC/Kalah-Framework/issues/7`

was set directly and was prone to concurrency issues if an agent decided to start more than one thread.

A protocol would solve issues like these by decoupling the specific details of how client is implemented (language, execution method, ...) from the framework. This is possible if a well-defined communication interface does is designed to consciously avoid any assumptions on how a client — or a server — is implemented.

At the same time it must be kept in mind that the course is focused on artificial intelligence, not systems programming, network communication or the issues of parsing and interpreting a language. The protocol must hence strike a balance between ensuring flexibility for the creative and eager students, without overburdening or annoying the others.

2.2 The Protocol

The protocol was designed in private correspondence between Tobias Völk and myself from January 2021 until November of the same year. The results were formalized in a specification, which is reproduced in appendix A. It was given the name "Kalah Game Protocol" (KGP).

KGP is a line-oriented, plain-text protocol, comparable to UCI and GTP. The communication is asymmetrical, and it is expected that clients connect to a common, well-known server. It is therefore only necessary for an agent to implement the client-side part of the protocol. This fact is used to leverage complexity towards the server whenever possible, so that implementing an agent can be kept as simple as possible.

2.2.1 Version Agreement

After a client has established a connection to a server, by whatever means available, the server sends out a single line establishing the protocol and the current revision of the protocol

```
kgp 1 0 0
```

The three arguments to the command `kgp` are the major, minor and patch versions. These indicate backwards-incompatible, forward-compatible and simple bug fixing changes respectively, a practice adapted from "semantic versioning".

If the client is not familiar with the stated iteration of the protocol, it is expected to not proceed any further and terminate the connection.

2.2.2 Protocol Modes

Otherwise the client may respond by signalling what functionality it is interested in. This is done by requesting an protocol "mode". As of writing, the main such mode is called `freeplay`. The client would request it as follows:

```
mode freeplay
```

Having selected this specific mode, the client will be sent a series of challenges that it should respond to. Each challenge consists of a state of the board, that is not required to bear any relation to any previous challenge

```
state <3,1,3,0,4,4,4,3,3>
```

The client responds by sending a `move`-command with one numerical argument, indicating the move it believes should be made:

```
mode 2
```

The client is allowed to update this decision as often as it wants, up until the server sends out a

```
stop
```

command. Any update received after this point is silently ignored. The `stop`-command is usually sent out when the time allotted for a decision to be made has run out. If the client is interested in moving things ahead more quickly, it may indicate that it does not intend to change its mind by sending out

```
yield
```

after which another `state` request may be issues. This cycle has been summarized in figure 1.

2.2.3 Command IDs and References

It is specified that in `freeplay`-mode the server is allowed to send out as many `state`-commands as it decides to. For this to work, the client must be able to make explicit `state`-command it is referring to when responding.

This is done by an optional system of command IDs and command references. Each command may be prefixed with neither, either nor both, as seen in the following examples:

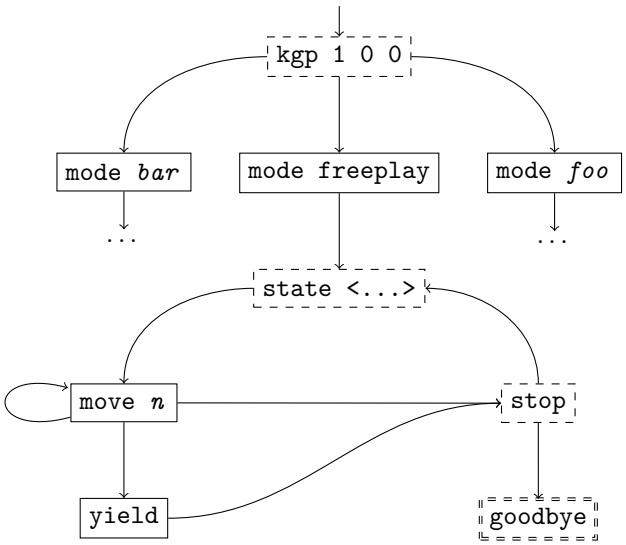

Figure 1: State-diagram of `freeplay`. The dashed boxes indicate messages sent out by the server, the non-dashed boxes indicate messages sent out by the client.

```
1 foo
@2 bar
3@1 baz
```

The command `foo` has the ID 1, `bar` refers to some command with the ID 2 and `baz` has the ID 3 while referring to the command with the ID 1 (`foo`).

The semantics of a valid reference is up to the active mode. "Valid" means that there exists an unambiguous reference between a unique and actually existing command with some ID i and a subsequent command that refers to i.

As implied above, in `freeplay` references are used to avoid concurrency issues w.r.t. `move`, `yield` and `stop` when dealing with multiple concurrent `state`-queries. That being said, it is also useful, even if the server does not intend to send out concurrent `state`-queries, as it prevents race conditions when the server decides to `stop` a `state`-request and immediately sends out a new one, while the client sends a `move` command. Without a proper reference, the server wouldn't be able to decide with certainty if the `move` command was supposed to refer to the previous or the current `state` request. In this worse case this might lead to an unintentional move being made. Due to network lag, this issue arises more often than one would expect.

2.2.4 The `set`-Command

As argued in section 1.2.3, it is inherently preferable to have a flexible and forward-compatible protocol. The lack of flexibility to address changing requirements will inevitably doom a system to a premature death.

Yet reliance on too much flexibility may lead to fragmentation and make implementing a protocol more complicated than need be the case.

The approach taken by KGP was previously mentioned under the term "weak extensibility". In our context this will specifically mean that both client and server can send messages, that can, but explicitly *must not* be handled by the respective opposite side. That is to say that none of the core functionality of a mode may suffer from either side not supporting the "weakly" extended part of the protocol.

This approach is done by introducing a separate command called `set`. Superficially it is intended to communicate a shared state, such as when the client intends to inform the server of the author's name or when the server wants to make clear what the name of the implementation is. The former example may look like this:

```
set info:author "John Doe"
```

The command takes two arguments, an option name and a value. All option names are delimited

by a colon (`:`) that separate the actual option name from the "group" it is part of. The specification currently defines three such groups (`info`, `time` and `auth`). The intention is that while handling each group is optional, an implementation is urged to support all options in a group.

2.3 Rationale

As mention at the beginning of the section, we would like to reflect upon the sketch of the protocol outlined in section 2.2 with the considerations elaborated in section 1.2.

The syntax follows the traditional convention of game protocols, in being plain-text and line-oriented. This decision was mainly made to avoid a dependence on an external parser or the need for students to implement one themselves, in addition to the protocol logic.

This point could be contended: Depending on the serialization format, most languages either provide libraries in their standard library or as packages that are distributed in some repository. The issue here is that no matter what format one decides to use (XML, JSON, S-Expressions,...), one would always disadvantage some language that either has no support in the standard library or makes bundling and distributing dependencies cumbersome.

Meanwhile, the entire syntax that KGP uses for `freeplay` could be parsed using a single regular expression.

On the question of connection-persistence, it might seem as though the decision has been made to make KGP connections persistent. This is not necessarily the case, but is instead a specific feature of the `freeplay`-mode. It is imaginable to have a separate mode for non-persistent connections:

```
kgp 1 0 0
mode oneshot
id 782593
state <3,1,3,0,4,4,4,3,3>
move 2
[ End of connection ]
```

where `id` would be a client-side command that identifies the agent in a game. If the need arose, the protocol specification could be updated to add support for such a mode, while at the same time a server would retain support for both `freeplay` and the hypothetical "`oneshot`" mode.

For now, it appears as though the persistent `freeplay`-mode is sufficiently well suited for the needs of KGP. One major advantage of a persistent connection is that it does a better job of abstracting away the transport layer. One can write a client by just operating on the standard output and input, then use a utility like Netcat to wrap the I/O and connect to a server that may accept connections using TCP. Similar tools exist for other transport layers such as WebSocket.

To summarize: The core protocol consists of give commands (`kgp`, `mode`, `set`, `ok`, used for generic confirmations, and `goodbye`). Functionality can be extended in two dimensions, either by defining a new mode or by defining a new command group.

It is the contention of the KGP developers that the protocol manages to find a successful and satisfying compromise between these opposing priorities.

3 Server and Client implementations

On a technical level it might be possible to require all participants in the tournament to implement their own protocol clients, but considering the disproportional popularity of certain languages (e.g. Python, Java), it makes sense to provide KGP client-libraries for these languages.

In this section we will comment on both the currently available clients and the server used in the tournament.

3.1 Clients

As of writing, there exist three client implementations: Two libraries for Python, Java and a program that simplifies implementing a client in a custom language.

3.1.1 Python Client

A rather straightforward implementation of KGP client-side protocol is the python client, simply called `pykgp`. It is distributed in a single file and is distributed in the KWARC KGP repository[6].

The source code for a simple client is given in figure 2. This is under the assumption that the library

[6] `https://github.com/KWARC/kalah-game/`

```
import kgp
import random

def random_agent(state):
  moves = state.legal_moves(kgp.SOUTH)
  yield random.choice(moves)

kgp.connect(random_agent)
```

Figure 2: A minimal, random-move client written in Python.

file (`kgp.py`) is in either the Python library path or in the current working directory.

The library implements `freeplay`-mode for both TCP and WebSocket connections. For each `state`-command is handled using the `multiprocessing` module, that is to say that a new process is spawned and killed as soon as the query is aborted via a `stop`-command.

3.1.2 Java Client

The Java library was principally designed and implemented by Tobias Völk, a fellow student at the FAU. It is called `jkgp` and is distributed using a `.jar` archive.

Just like with Python, it implements the `freeplay` mode. Instead of handling each request in a separate process, `jkgp` does so using separate threads. Among other things, this also implies that due to Java's threading model a thread cannot be terminated from outside its own execution. Instead, the agent is required to manually check if a `stop`-command has been received and terminate the thread when it chooses to do so.

3.1.3 Wrapper client

A different approach to the previous libraries is a client that further simplifies the process of using whatever language one may be interested in. The standalone client, called `kgpc`, might be started like in the following:

```
$ ./kgpc my-client some-server.net:2671
```

where `my-client` is an executable file and `some-server.net` would be serving a KGP service on the port 2671.

Instead of, as in the previous examples, implementing KGP from the bottom up, starting from a transport layer like TCP or WebSockets, a generic client would wrap an executable. Similarly to `pykgp`, each `state`-command would have a new process created and then execute the executable with is passed as a command line argument. This new process would then receive a board state via the standard input,

```
3
1 3
0 4 4
4 3 3
```

and `kgpc` would expect a response via the standard output:

```
2
```

When the time has run out (as indicated by a `stop`-command) the child process would be killed.

This client, called `kgpc` was implemented in Go.

3.2 Server

There currently exists only one serious server implementation, called `go-kgp`. As the name implies, it was written using the Go Programming Language. Like all the above, the source code is also distributed in the KWARC repository.

The server has two operational modes, the first just listening for incoming connections and then starting games with whatever clients are available. If clients authenticate themselves (this is done by setting the `auth:token` variable, see section 4.2), the performance of agents is kept track of in a scoreboard.

The second mode provides a controlled "tournament" environment, where instead of providing a public service that anyone may connect to, the server manages the clients directly. A client can be configured to be isolated within a container, thus also allowing the resources to be limited. Internally this mode is referred to "*Kalah*seum".

`Go-kgp` uses SQLite to store all the data generated during matches.

Currently the server still has potential of being improved and stabilized. In particular the latter mode

still suffers from difficulties with certain isolation mechanisms like Docker.

It would be desirable to have multiple server implementation in the long term, perhaps even splitting the two operational modes into separate projects.

4 Technical Evaluation of a Tournament

This section will recapitulate the development and execution of the first KGP-based tournament that took place in the winter semester of 2021 at the Friedrich-Alexander University (FAU) in Erlangen.

With the server ready and client libraries available, the tournament began on 2021-12-02. The tournament was divided into two stages

Open, Training Tournament Up until the submission deadline (2022-01-09) a publicly accessible server was provided for clients to connect too anonymously. If the students want to add a pseudo-anonymous nickname to their client, they could do so too. This server would randomly associate waiting clients and have them play a match between themselves.

The result of this match would be noted and an ELO score would be updated on a public scoreboard. The placement in this scoreboard had no effect on the final grade.

Closed, Tournament After the submission deadline, the second phase of the tournament was initiated where all submitted agent implementations were pitted against one another in controlled, isolated environments with the same available resources (CPU cores, memory, storage). The tournament would then take place in multiple rounds, simultaneously eliminating those agents that performed less well and increasing the complexity of the game.

All agents that managed to pass a "smoke test" involving a game with a random client, received 100 points, and the top 10 participants received 100 points, 90 points, 80 points, etc. respectively. The points allotted in this tournament were bonus points for the course exercise, that in turn entitled the student to bonus points in the final exam.

This division mirrors the operational modes described in section 3.2.

The open tournament phase was useful to detect bugs in both `jkgp`, `pykpg` and the server and have them fixed in time for the closed tournament.

4.1 Transport Layer

Prior to this, an initial version of `go-kgp` had been developed as a proof of concept for concept of using a protocol instead of a single-language framework as described in section 2.1. This version had initially only support for TCP connections. After discussions with the KWARC team (in particular Tom Wiesing) it was concluded that TCP could not be used, since the university network and its administrators had made it difficult to open a regular port. As an alternative, it was decided to use WebSocket Protocol [4], that allows for reliable, bidirectional communication on top of HTTP. The cost was a complication in both the client libraries and the server, the advantage is that communication via WebSockets can make use of HTTPS and thus provide encrypted connections.

4.2 Authentication

Around this time an ongoing discussion was that of authentication. While KGP is inherently anonymous, tracking agent identities was of use for the open tournament mode, so that a scoreboard could be created. The considered two options were "challenge-" or "token based" authentication.

The former would involve a cryptographic challenge, to prove that a client has access to a secret. Assuming the client were to provide an ID (which might be equal or derived from a public key)

```
set auth:id "3d2ac57e788abf"
```

the server would respond with a challenge:

```
set auto:challenge "7687fee867bdad"
```

to which the client would have to provide the right response:

```
set auto:response "44d12a26ee3dbb"
```

Only if these actions are preformed in this order, and the server is satisfied by the result, would the connection reliably associated with an ID.

Implementation	Count
Python	16
Java	13
C++ and Python	1
C and Python	1
Kotlin	1

Table 1: Number of concrete implementations per programming language

Token based authentication is, compared to the previous suggestion a simple affair, in that all it requires is that the client sends the server a secret:

```
set auth:token "Any string you like."
```

Anyone who provides the same token is assumed to be the same entity. Of course, for this to be secure, one would have to rely on the connection being secure.

As authentication was only of interest for the public server, that relies on of an encrypted WebSocket connection as discussed above, the latter method was chosen for the sake of simplicity, and to avoid having to make the students have to "roll their own crypo".

4.3 The `simple-mode`

During the developmental phase, a second protocol mode was specified and initially implemented by both `jkgp` and `go-ktp`. The mode was a variation on `freeplay` and was given the name `simple`. It was meant to avoid the use of reference numbers by requiring each `stop` to be confirmed by a `yield` response. The server would keep track of the difference and only accept a `move` if the difference was 0. If the difference was positive (outstanding `yield`, at least once `stop` was sent out but fewer `yield`s were received), any response would be ignored. If negative, the connection would be terminated as the client would be regarded to have entered an illegal state. Due to its nature, simple mode could only be used to play one game at a time.

It was concluded that despite the name, `simple-mode` was more complicated that `freeplay`. The advantage of not having to track command IDs and references was relatively marginal. It is therefore planned to deprecate the mode in the future.

4.4 Evaluation of the Closed Tournament

For the closed tournament, there were some initial difficulties with making sure that each submitted client could be built and managed by the server. These had to be resolved on an individual basis. In table 1 an overview is given of how common different languages were. As can be seen, in most cases either `pykgp` or `jkgp` was used, where in one case `jkgp` was used together with Kotlin, an alternative JVM-based language. There were two attempts at implementing the performance-critical components of an agent in C and C++. These did not re-implement the protocol from the ground up, instead relying on `pykgp` by invoking the C components via a Python shim.

There were a number of submissions that could not be fixed or were not prepared properly. In a few cases there appeared to be a misunderstanding, with programs being submitted that had no relation to KGP. Instead, these just implemented Kalah as a standalone library or application.

In other cases the programs were submitted with invalid Docker specifications (`Dockerfile`), and could thus not be tested. A few plagiarisms were detected which were disqualified..

Setting aside these unfortunate cases, the project was successfully executed, in both the open and close phase. KGP should be able to serve as a flexible and powerful basis for future tournaments.

5 Future Thoughts and Plans

While the organization of the tournament may be regarded as a success, there remain issues we observed that ought to be addressed and ideas that have become apparent in retrospect that deserve further thought. In this section, we would like to conclude this report by dwelling on these points to improve future tournaments using the *Kalah Game Protocol*.

A few points of consideration would include:

- Providing more libraries for multiple languages. Languages of interest might be C/C++, Prolog, Common Lisp, Rust and/or Julia.

- Improving the specification to address underspecified parts of the protocol. Issues that

should be considered could be setting a maximum line length, what encoding the protocol should use and error handling.

- The "open tournament" server should be improved in the amount of information it has to offer, and the methodology used in scoreboard system.

  The current system makes use of the ELO rating system. Paring was done by having agents of similar scores compete with one another. This system would work optimally if all agents were active at all times. As this is not a legitimate assumption, the rating and pairing system should accommodate this fact.

- To avoid issues that might arise during the submission phase, all libraries should be distributed within *ready-made* templates that would include `Dockerfile`s. These would be directly suited for development and could contain instructions and utility scripts for testing and preparing a submission.

- Rating winners in multiple disciplines, besides just "overall performance". This might include speed, the ability to deal with difficult edge-cases or resource efficiency.

Bibliography

References

[1] Daniel Bump, Gunnar Farneback, and Arend Bayer. *GNU Go Documentation*. 2009.

[2] W Eddy. "RFC 9293 Transmission Control Protocol (TCP)". In: (2022).

[3] Gunnar Farnebäck. *Specification of the Go Text Protocol, version 2, draft 2*. 2002.

[4] Ian Fette and Alexey Melnikov. *The websocket protocol*. Tech. rep. 2011.

[5] Michael Genesereth, Nathaniel Love, and Barney Pell. "General game playing: Overview of the AAAI competition". In: *AI magazine* 26.2 (2005), pp. 62–62.

[6] Michael R Genesereth, Richard E Fikes, et al. "Knowledge interchange format-version 3.0: reference manual". In: (1992).

[7] Stefan-Meyer Kahlen. *Description of the universal chess interface*. Apr. 2004. URL: `http://wbec-ridderkerk.nl/html/UCIProtocol.html`.

[8] Nathaniel Love et al. *General Game Playing: Game Description Language Specification*. Tech. rep. Stanford University, 353 Serra Mall, Stanford, CA 94305: Stanford Logic Group, 2008.

[9] David Maier et al. "Datalog: concepts, history, and outlook". In: *Declarative Logic Programming: Theory, Systems, and Applications*. 2018, pp. 3–100.

[10] Tim Mann. *Frank Quisinsky interviews Tim Mann about XBoard and WinBoard*. 2000. URL: `https://tim-mann.org/history.htmlpr`.

[11] John McCarthy. "Recursive functions of symbolic expressions and their computation by machine, part I". In: *Communications of the ACM* 3.4 (1960), pp. 184–195.

[12] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), pp. 484–489.

[13] H.G.Muller Tim Mann. *Chess Engine Communication Protocol*. URL: `https://www.gnu.org/software/xboard/engine-intf.html`.

[14] Tobias Völk. "Evaluation of Performance and Techniques used in Kalah Competition". In: (2022).

[15] Bruce Wilcox. *Standard Go Modem Protocol - Revision 1.0*.

A The "Kalah Game Protocol" Specification

The following pages include a copy of the current state of the KGP specification (Version 1.0.0), refereed to throughout this document.

As mentioned in section 5, there are still ambiguities that ought to be clarified.

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 1.0.0 of the KGP protocol.

Contents

1 Prelude **1**
- 1.1 Definitions . . . 1
- 1.2 Formal Structure . . . 1
- 1.3 Protocol Overview . . . 2

2 Defaut Modes **2**
- 2.1 Freeplay Mode . . . 3

3 Freeplay commands **3**

4 Simple Mode **3**
- 4.1 Simple commands . . . 3
- 4.2 Examples . . . 4
- 4.3 Evaluation Mode . . . 4
- 4.4 Evaluation commands . . . 4

5 Responses **4**

6 The `set` Command **4**
- 6.1 `info`-group . . . 5
- 6.2 `time`-group . . . 5
- 6.3 `auth`-group . . . 5

7 Notes **5**

8 Distribution of This Document **5**

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 0
```

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 0
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`.

4 Simple Mode

The "simple" mode restricts "freeplay" by introducing a more strict state model, thus relieving both client and server from having to track IDs. The main intention is to make client implementations easier, by shifting the burden of synchronisation management on to the server.

As such "simple" mode is convenient for implementing tournaments agents.

4.1 Simple commands

`state [board] / stop` (server) The server MUST alternate `state` and `stop` commands (possibly, with other commands inbetween of course), starting with a state command.

`yield` (client) The client MUST send `yield` after it has finished searching, no matter the reason. The client MUST not send `yield` in any other situation.

There are three cases:

- The server sends `stop`, the client replies with `yield`
- The client sends `yield`, the server replies with `stop`
- The client sends `yield` "at the same time" as the server sends `stop`, neither replies to the other (they already did)

4.2 Examples

Example of a slow client (unrealistically short game):

```
s: kgp 1 0 0
c: mode simple
s: state <4,0,0,3,0,0,0,1,1,1,1>
c: move 1
s: stop
s: state <3,1,0,0,4,4,3,3,3>
c: move 1
c: move 1
c: yield
c: move 2
c: move 4
c: move 3
s: stop
c: mode 4
c: yield
s: goodbye
```

In this example the client did not realize the search period was ended and keeps sending moves for the old state even though the server has (rightfully so) already sent the next state query.

Example of an impatient server/slow client (unrealistically short game):

```
s: kgp 1 0 0
c: mode simple
s: state <...>
s: stop
s: state <...>
s: stop
s: state <...>
s: stop
c: yield
c: yield
c: yield
s: state <...>
s: stop
c: move 1
s: state <...>
s: stop
s: state <...>
s: stop
c: yield
c: move 2
c: move 2
c: yield
c: move 3
c: yield
s: goodbye
```

4.3 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.

4.4 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.

5 Responses

6 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:

6.1 `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, ...

6.2 `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.

6.3 `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.

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`.

7 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.

8 Distribution of This Document