The OthelloZoo manual

Stéphane Nicolet <cassio@free.fr>

Draft 0.3 (22 December 2010)

Contents

1 Introduction

Wouldn't it be nice if we could use the power of remote computers to play othello? There are thousands of idle computers out there on the Internet, and none of them is used for doing really interesting and funny stuff, like solving othello endgames faster, and the like. This document is the description of such a grid system, called the OthelloZoo.

2 Architecture

The OthelloZoo is organized around an abstract database of othello positions which serves as a shared memory for othello clients. Programs can post positions to the zoo, seeking help from other othello clients to solve the positions they posts, and retrieve the results once the positions are calculated.

Who could use that ?

I started the project with the aim of writing a massively parallel version of Cassio, a quite complete othello program running on Macintosh, to speed up its endgame calculation by a factor of ten. After the idea of the initial project was presented to the othello community, other usage patterns appeared to use the grid concept for the game : cracking or solving the game [Jan de Graaf]; make better internet clients for studying the transition between openings and endgame [Michele Borassi].

What kind of programs can connect to the zoo ?

There are at least three natural kinds of programs:

- Pure consumers: A consumer program will just post jobs to the zoo and listen for the answers, without sharing their computing power. Such a client could be used, for instance, in PDA or iPhone othello program to accelerate endgame calculations, as the processors of these machines typically calculate an order of magnitude slower than usual personal computers. Another example of a pure consumer program would be a meta-manager, which distribute subtasks in parallel to the zoo with the ultimate aim of solving the game.
- Othello daemons: on the other hand, othello daemons are very altruist and share their CPU cycles for the zoo. They are small utility programs or screen savers running on personal computers, which continuously poll the zoo to get a position to solve, calculate it and send the result back to the zoo. They should be written using a fast othello solver, and operate in the background as transparently as possible on the computer on which they run (on idle user time, or with a low CPU priority).
- Interactive clients: possibly the most exciting use of the zoo (but also the most difficult to implement for program authors) would be to adapt existing othello programs to the zoo. The adaptation could be two ways : first, they could use the idle user time (I mean as soon as the othello engine is not used, for instance when the user is pondering on his/her next move or browsing the Wthor database) to switch to daemon mode and contribute (transparently for the user) to the zoo; secondly, it is possible to parallelize the alpha-beta algorithm used in endgame search, using the zoo system to distribute the subtree calculations, to speed up the user experience time when he most needs it.

How much of the total CPU power of the zoo will I get?

That question simplifies to: how can we be sure that the zoo system is fair between its users? It is a tough task to find a good load balancing strategy when dispatching jobs. For instance, we don't want one master abusing resources of the zoo by sending too many jobs at the same time, or too difficult jobs (=too long to solve, with too many empty squares), which would lead to starvation of the other users of the zoo. One idea (not implemented yet) is to record the total CPU time that each consumer has got in the last 20 minutes or so and affect jobs to different masters using some sort of time balancing strategy.

Another point is the length of the jobs : as always with parallelism, too small and too large chunks of work are inefficient.

- *too small* : trivial positions would typically waste everybody's time because they could be solved faster by the local program instead of using the network.

- *too large* : apart from the load balancing problem, there is the difficulty that the zoo is, in a sense, unreliable : be they interactive clients (customized versions of WZebra or Cassio), daemons or screensavers, calculators can cancel their calculations at any time (and we have to handle network failure as well), so we probably shouldn't launch long endgame solves because there would always be a possibility of wasting resources.

What operating system does the zoo run on?

The zoo database is designed to be accessible with standard Internet protocols, so that clients on different computers and different operating systems can post and send results to the zoo using classic network technologies (telnet, sockets, HTTP request, etc.).

Security and redundancy

It is sometimes necessary, or at least benefic, to be able to double check the calculations made on a grid by remote computers. It should be possible to send the same position twice, so that the same position is solved by different calculators for the sake of verification.

Remark: this is achievable by the prototype of the zoo, which has a remanence of 15 minutes at the moment. Since the positions are only kept for 15 minutes, it is possible to post jobs twice with a 30 minutes interval and get the endgame solves double-checked.

3 Getting connected to the zoo

3.1 Connecting

The test server for the OthelloZoo is accessible at the IP address `82.230.184.124` on port 80. There is no authentification step: as soon as a client has established a connection, it can start to send commands using the OthelloZoo protocol.

Since opening and closing a connection takes time and consume resources of the server, the preferred way to establish and use a connection to the OthelloZoo is via a permanent connection (as implemented in telnet programs, for instance), using BSD sockets (or Windows sockets).

Remark: At the time of this writing, the address of the OthelloZoo server is fixed (because the test server is running on my personal computer). Maybe we should implement some sort of publication scheme for the server address. The approach used by the BOINC project for distributed computing is interesting here, because it uses one level of redirection and allows to adjust the number of servers, if necessary: they advocate putting a page in a fixed position on the Web with special XML tags ("scheduler"), each such tag containing the address of an active grid server; each time a client wants to join the grid, he first downloads and parses this web page, gets the list of the active servers for the grid project, and can connect to the grid on one of these servers (see http://boinc.berkeley.edu/trac/wiki/ServerComponents#ThemasterURL).

3.2 Disconnecting

During a session, the server will disconnect any client which has not shown a sign of activity for 30 seconds. As a consequence, to avoid being disconnect, clients should send a simple **KEEP_ALIVE** command to the server every 15 seconds.

At the end of a session, clients must send a **STOP_ALL** command (for a consumer client) or a **I_QUIT** command (for a calculator client) to the server. This will allow the server to take appropriate measures and to release cleanly the resources the clients had taken. As soon as a client has sent its quitting command on the network, it can close its half of the connection: there is no acknowledgement phase.

3.3 The zoo server

The prototype of the OthelloZoo server was written as a PHP script connecting to a mySQL database, but was suffering from charge congestion under heavy load. The current server is written completely in Java, using the Apache MINA library from the Apache Foundation for the network part, and the H2 Database Engine library for the implementation of the SQL database.

The Java language was chosen because it seems to provide the right balance between speed, reliability and portability : the jar file of the server can be executed on any computer with a java machine, while the chosen libraries allows us to handle connections and database queries to the zoo without any file access (everything is in memory).

3.4 Example code

[Stéphane: a portable wrapper around zebra, written in C with **libcurl** for the networking stuff ?]
[Jan: a wrapper around zebra in C#?]

4 The OthelloZoo protocol

Once they have established a connection to the zoo, clients can start sending requests to the database. Each request consists of a single line command, and the zoo sends back an answer consisting of one or more lines. The protocol is asynchronous and acknowledgement-free : clients don't have to wait for the results of a previous command before sending another command, and there is no acknowledgement phase.

4.1 Command syntax : how to send action parameters to the zoo?

In this document, the description of commands will be typeset with the following typographical presentation:

COMMAND-NAME ⟨*parameter-name*⟩ ⟨*parameter-name*⟩ ⟨*parameter-name*⟩

For instance, one of the most useful command is the **GET_WORK** action, which asks the zoo to give us a job, that is, a position to solve (it will be described in more details later in the manual). This command will be typeset in this document as:

GET_WORK ⟨*asker*⟩

However, when transmitting commands to the zoo database, you should use a URL-like syntax, with the command name following the keyword `?action=` , and the parameters separated by the ampersand character `&`. This means, for instance, that if you want, in your program, to use the **GET_WORK** command above with a value of 1234 for the ⟨*asker*⟩ parameter, you will have to send the following query to the zoo through the internet, with a linefeed character (ASCII `#0A`) to terminate the line:

```
?action=GET_WORK&asker=1234
```

Note that the 'action' keyword containing the command name should always appear first, but that the order of the other parameters is free. For instance, these two lines represent the same query to the zoo :

```
?action=GET_RESULTS&asker=333555777&date=120053636
?action=GET_RESULTS&date=120053636&asker=333555777
```

4.2 List of commands

This is the minimal list of commands to implement a master program, and their typical use:

ADD	The master sends a job to the zoo. This action can be repeated to delegate simultaneously several different jobs to the zoo.
GET_RESULTS	By polling the zoo every second, the master listens to the eventual new results of his jobs (since a given date, chosen by the master).
STOP	At any moment, the master can stop (cancel) a number of jobs he had previously sent to the zoo. All current calculators working on these jobs will stop.
STOP_ALL	The master stops (cancels) all the jobs he had previously sent to the zoo. This command should be sent by a master program when he quits.

This is the minimal list of commands to implement a calculator program, and their typical use:

GET_WORK	The calculator asks for work, and gets a job in return from the zoo.
STILL_NEEDED	The calculator polls the zoo (every second) to know if the job he is calculating is still needed. If it is no longer needed, the calculator stops its current calculation and can then ask a new job to the zoo.
I_CANT_SOLVE	The calculator warns the zoo that he can't finish a job, so that the zoo may decide to give the job to someone else.
SEND_SCORE	The calculator sends back the score of a finished job to the zoo.
I_QUIT	The calculator warns the zoo that he must quit.

Some sequences of commands happen repeatingly when writing and using a zoo client (either master or calculator), because they correspond to common pattern usages of a program logic. To use the network more efficiently, the OthelloZoo protocol proposes the following combined commands, which allows clients to emulate the effects of successive commands in a single transaction:

ADD_AND_GET_RESULTS	combination of **ADD** and **GET_RESULTS**
STOP_AND_GET_RESULTS	combination of **STOP** and **GET_RESULTS**
ASKER_TAKES_IT_AND_GET_RESULTS	combination of **ASKER_TAKES_IT** and **GET_RESULTS**
GET_WORK_AND_GET_RESULTS	combination of **GET_WORK** and **GET_RESULTS**
STILL_NEEDED_AND_GET_RESULTS	combination of **STILL_NEEDED** and **GET_RESULTS**
SEND_SCORE_AND_GET_WORK	combination of **SEND_SCORE** and **GET_WORK**

The following two commands can be used to check the network and keep permanent connections alive. They are useful both for master and calculator clients.

PING	pings the server to know if the othello zoo is available
KEEP_ALIVE	asks the server not to close the connection; this command should be issued at intervals of about 15 seconds, or 15 seconds after the last command to the server.

Special commands (advanced topics and debugging):

ASKER_TAKES_IT	master warns the zoo that he decides to calculate himself one of the jobs he had added. A calculator working on that job is *not* interrupted.
CHECK_PREFETCH	calculator verifies the usefulness of pipelined jobs (cf pipelining, [ref needed]).
VIEW	human wants to see the OthelloZoo database in a browser.

4.3 Results set : how to read the results of the zoo?

We have seen that each command is sent as a single line of text to the server. The server will send back its answer as a stream of one or several lines of text separated by a linefeed character (ASCII `#0A`). The protocol is asynchronous : clients don't have to wait for the results of a previous command before sending another command (in other words, there is no acknowledgment phase in the protocol).

The results sent back by the zoo for each transaction are basically a set of positions with their properties, each position in the set being prefixed by a status keyword telling the client how it should handle the position.

The list of status keyword in response to the **GET_RESULTS** command is the following :

CALCULATED : this keyword indicates that the submitted position has been solved, and gives back the result. This is good news !

INCHARGE : this keyword indicates that the zoo has managed to dispatch the position to a calculator, which has taken the responsibility to solve it.

COULDNTSOLVE : this keyword indicates that the calculator which had previously taken the position to solve it has released the position to the zoo. The zoo will now try to give the position to another calculator.

DELETED : this keyword indicates that the position has disappeared from the zoo (because the master canceled the job).

PREFETCHED : this keyword indicates that the position has been prefetched to a calculator by the zoo (see chap. pipelining).

4.4 An example of session

Here is an example of a telnet session showing some of the basic commands (**PING** and **GET_WORK**) and their results, when the database is empty. Before we issue the second **GET_WORK** command, another program has connected to the zoo and added some positions for solve in the database, so that the second **GET_WORK** gives us a job to solve. Note the syntax of the results, where results fields are separated by spaces. Finally, the last line shows that the zoo server has disconnected us automatically after 30 seconds of inactivity.

```
telnet 82.230.184.124  80

   Trying 82.230.184.124...
   Connected to 82.230.184.124.
   Escape character is '^]'.

?action=PING
   PING ANSWERED
   END.

?action=GET_WORK&asker=24353646
   NO JOB !
   NO PREFETCH !
   END.

?action=VIEW
   <HTML><HEAD></HEAD><BODY>DATABASE IS EMPTY</BODY></HTML>
   END.

?action=GET_WORK&asker=24353646
   JOB pos=O-OOOOOOOOOOXXOOOOXXOXOOXOXOXXOOXXOOOXOOOOXOXOO-OOOXX-O-XXXXX--X window=-64,64 cut=95 depth=6 hash=60529449
   PREFETCH pos=O-OOOOOOOOOOXXOOOOXXOXOOXOXOXXOOXXOOOXOOOOXOXOO-OOOXX-O-XXXXX--X window=-7,6 cut=100 depth=6 hash=1433001
   END.

   Connection closed by foreign host.
```

5 Commands reference

5.1 Parameters common to many commands

The OthelloZoo protocol is all about sending positions from one othello program to another, and information about these positions. As a consequence, it is no surprise that a lot of the commands share a small set of common parameters. This list of these common parameters is explained here (note that we will explain the specific parameters for other commands when they appear):

⟨**asker**⟩ : a unique (31 bits, positive) integer, identifying for the zoo the computer and the instance of the program which sends the command. This should be unique to a session : if the same client runs on two different computers at the same time (or on two different cores on the same computer), each should connect to the zoo using a different ⟨*asker*⟩ parameter. One way to achieve this is to use a random generator at program launch time to generate the number, with the random seed carefully initiated with the sum of the process serial number of the client and the number of microseconds since the start of the computer.
Remark : discuss the possibility to ask the server for a valid ID at connection time? Less time efficient, but could lead to less collisions; could also lead to a false sense of security in case of race conditions.
Example: `&asker=31415926`

⟨**pos**⟩ : a 65-character string description of an othello position. The first 64 characters contain the contents of the squares in row-wise order, *ie* a1-b1-c1-...-g8-h8, while the 65th character indicates whose turn it is to move. Black discs are denoted by X, white discs are denoted by 0 and empty squares are denoted by -.
Example: `&pos=--------------------------OX------XO--------------------------X`

⟨**depth**⟩ : the number of empty squares in the position.
Example: `&depth=25`

⟨**window**⟩ : the [alpha,beta] window of the required solve of the position. Alpha and beta can be any integers in the range $-64..64$, provided that $-64 \leq \alpha < \beta \leq 64$. For instance, a complete solve will be $[-64, 64]$, a win/loss/draw solve will be [-1,+1], while a [6,7] solve checks if the position is winning by at least +8.
Example: `&window=6,7`

⟨**cut**⟩ : the probcut selectivity level for the required solve of the position, denoted as an integer in the range 0..100. A value of 100 means a result sure at 100%, while a value of 0% means 'mostly unreliable'. The proposed sequence of levels is a subset of the levels used in WZebra: $57, 72, 83, 91, 95, 99, 100$. Since the sequence of probcut levels can vary among calculators, if calculators can't answer exactly at the required level of selectivity, they should use the *next* level of probcut in their internal sequence to calculate the answer of the job.
Example: `&cut=83` note : the % sign is NOT included.
`&cut=100` note : the % sign is NOT included.

⟨**hash**⟩ : a unique (31 bits, positive) hash value, identifying a job sent to the zoo.
This hash value should (a) be *unique* for each $(pos, depth, window, cut)$ quadruplet, so that a master can send, for instance, the same position to solve with different alpha-beta windows or different selectivity levels: the different alpha-beta windows and different selectivity levels will make different jobs; (b) be *uniformally distributed* in the range 1..2147483647. The proper way to get these two properties is to use the 31 least significant bits of a (good) hash function on the text string $(pos + depth + window + cut)$.
We recommand to take the absolute value of the Jenkins hash function given in the appendix.

5.2 Basic master actions

5.2.1 the ADD action

Syntax: **ADD** ⟨*pos*⟩ ⟨*window*⟩ ⟨*cut*⟩ ⟨*depth*⟩ ⟨*hash*⟩ ⟨*priority*⟩ ⟨*asker*⟩

Usage: master sends a job to the zoo.

5.2.2 the STOP action

Syntax: **STOP** ⟨*asker*⟩ ⟨*hash*⟩ ⟨*h2*⟩ ⟨*h3*⟩ ⟨*h4*⟩ ⟨*h5*⟩ ⟨*h6*⟩ ⟨*h7*⟩ ⟨*h8*⟩ ⟨*h9*⟩ ⟨*h10*⟩

Usage: master stops (cancels) a number of jobs he had previously sent to the zoo (it is possible to stop up to 10 jobs with this command). Any potential calculators working on theses jobs will stop.

5.2.3 the STOP_ALL action

Syntax: **STOP_ALL** ⟨*asker*⟩

Usage: master stops and cancels all the jobs he had previously sent to the zoo. Should be sent by a master program when he quits.

5.2.4 the GET_RESULTS action

Syntax: **GET_RESULTS** ⟨*asker*⟩ ⟨*date*⟩

Usage: master listens to the eventual new results (since the given date) of the jobs he has sent to the zoo.

5.3 Basic calculator actions

5.3.1 the GET_WORK action

Syntax: **GET_WORK** ⟨*asker*⟩

Usage: calculator gets a job from the zoo.

5.3.2 the STILL_NEEDED action

Syntax: **STILL_NEEDED** ⟨*pos*⟩ ⟨*window*⟩ ⟨*cut*⟩ ⟨*depth*⟩ ⟨*hash*⟩ ⟨*asker*⟩

Usage: calculator asks the zoo if the job he is calculating is still needed.

5.3.3 the I_CANT_SOLVE action

Syntax: **I_CANT_SOLVE** ⟨*pos*⟩ ⟨*window*⟩ ⟨*cut*⟩ ⟨*depth*⟩ ⟨*hash*⟩ ⟨*asker*⟩

Usage: calculator warns the zoo that he can't finish a job.

5.3.4 the SEND_SCORE action

Syntax: SEND_SCORE ⟨*pos*⟩ ⟨*window*⟩ ⟨*cut*⟩ ⟨*depth*⟩ ⟨*hash*⟩ ⟨*asker*⟩ ⟨*score*⟩ ⟨*moves*⟩ ⟨*time*⟩

Usage: calculator sends back the score, optimal moves and time of a finished job to the zoo. The time parameter is the CPU time taken by the calculator to solve the position.

5.3.5 the I_QUIT action

Syntax: I_QUIT ⟨*asker*⟩

Usage: calculator warns the zoo that he quits.

5.4 Advanced actions

5.4.1 the ADD_AND_GET_RESULTS action

Syntax: ADD_AND_GET_RESULTS ⟨*pos*⟩ ⟨*window*⟩ ⟨*cut*⟩ ⟨*depth*⟩ ⟨*hash*⟩ ⟨*priority*⟩ ⟨*asker*⟩ ⟨*date*⟩

Usage: combination of ADD and GET_RESULTS. Note that the list of parameters, compared to the normal ADD action, has an added required date parameter, because the GET_RESULTS action takes a date parameter. This remark will also be true for the other combined actions : their set of parameters will be, most of the time, the union of the parameters of the actions they combine.

5.4.2 the STOP_AND_GET_RESULTS action

Syntax: STOP_AND_GET_RESULTS ⟨*hash*⟩ ⟨*asker*⟩ ⟨*date*⟩

Usage: combination of STOP and GET_RESULTS. In the current implmentation, this action can stop only one job at a time.

5.4.3 the ASKER_TAKES_IT action

Syntax: ASKER_TAKES_IT ⟨*asker*⟩ ⟨*hash*⟩

Usage: the master ⟨*asker*⟩ warns the zoo that he decides to calculate himself the job ⟨*hash*⟩ which he had previously added to the zoo. A calculator working on that job is *not* interrupted.

5.4.4 the ASKER_TAKES_IT_AND_GET_RESULTS action

Syntax: ASKER_TAKES_IT_AND_GET_RESULTS ⟨*asker*⟩ ⟨*hash*⟩ ⟨*date*⟩

Usage: combination of ASKER_TAKES_IT and GET_RESULTS

5.4.5 the GET_WORK_AND_GET_RESULTS action

Syntax: GET_WORK_AND_GET_RESULTS ⟨*asker*⟩ ⟨*date*⟩

Usage: combination of GET_WORK and GET_RESULTS. Useful to write a client that is at the same time a master and a calculator.

5.4.6 the STILL_NEEDED_AND_GET_RESULTS action

Syntax: STILL_NEEDED_AND_GET_RESULTS ⟨*pos*⟩ ⟨*window*⟩ ⟨*cut*⟩ ⟨*depth*⟩ ⟨*hash*⟩ ⟨*asker*⟩ ⟨*date*⟩

Usage: combination of STILL_NEEDED and GET_RESULTS

5.4.7 the SEND_SCORE_AND_GET_WORK action

Syntax: SEND_SCORE_AND_GET_WORK ⟨*pos*⟩ ⟨*window*⟩ ⟨*cut*⟩ ⟨*depth*⟩ ⟨*hash*⟩ ⟨*priority*⟩ ⟨*asker*⟩

Usage: combination of SEND_SCORE and GET_WORK

5.4.8 the PING action

Syntax: PING

Usage: pings the server to know if the othello zoo is available

5.4.9 the KEEP_ALIVE action

Syntax: KEEP_ALIVE ⟨*asker*⟩

Usage: asks the server not to close the connection; this command should be issued at intervals of about 15 seconds.

5.4.10 the CHECK_PREFETCH action

Syntax: CHECK_PREFETCH ⟨*asker*⟩ ⟨*hash*⟩ ⟨*h2*⟩ ⟨*h3*⟩ ⟨*h4*⟩ ⟨*h5*⟩ ⟨*h6*⟩ ⟨*h7*⟩ ⟨*h8*⟩ ⟨*h9*⟩ ⟨*h10*⟩

Usage (for pipelining): calculator verifies the usefulness of pipelined jobs (cf pipelining, [ref needed]). The hash descriptor of the first position (the ⟨*hash*⟩ parameter) is required, but the hash descriptors of the other positions (parameters ⟨*h2*⟩ to ⟨*h10*⟩) are optionnal.

5.4.11 the VIEW action

Syntax: VIEW

Exceptional usage: displays a web page with a dump of all the records of the SQL database implementing the OthelloZoo on the server. This is for debugging and monitoring only, as it tends to generate huge pages and could easily break the server down if used often. Programs should use the GET_RESULTS action and its variants to selectively poll for new results.

6 Appendix

6.1 Jenkins's hash function, in Pascal

```
// Hash function using Jenkins algorithm, see :
//    http://en.wikipedia.org/wiki/Hash_table
//    http://www.burtleburtle.net/bob/hash/doobs.html
//
function HashString(const s : String) : SInt32;
var hash : UInt32;        // unsigned 32 bits integer
    i,length : SInt32;    // signed 32 bits integer
begin

  hash := 0;
  length := LENGTH_OF_STRING(s);

  for i := 1 to length do
    begin
      hash := hash + ord(s[i]);          // ord(s[i]) is the ascii value of the i-th character
      hash := hash + (hash shl 10);      // shl is shift left
      hash := hash XOR (hash shr 6);     // shr is shift right
    end;
  hash := hash + (hash shl 3);
  hash := hash XOR (hash shr 11);
  hash := hash + (hash shl 15);

  HashString := SInt32(hash);
end;
```

6.2 Jenkins's hash function, in PHP

```
// Hashes a string using Jenkins algorithm, see :
//    http://en.wikipedia.org/wiki/Hash_table
//    http://www.burtleburtle.net/bob/hash/doobs.html
//
function my_hash_string($my_string) {
  $value = 0;
  $string_length = strlen($my_string);

  for ($i = 0; $i < $string_length ; $i++) {

     //var_dump($value);

     $value += ord($my_string[$i]);
     $value = ($value & 0xFFFFFFFF);

     $value += ($value << 10);
     $value = ($value & 0xFFFFFFFF);
```

```
      $value ^= ($value >> 6);
      $value = ($value & 0xFFFFFFFF);
    }

    $value += ($value << 3);
    $value = ($value & 0xFFFFFFFF);

    $value ^= ($value >> 11);
    $value = ($value & 0xFFFFFFFF);

    $value += ($value << 15);
    $value = ($value & 0xFFFFFFFF);

    // convert to 32 signed integer
    // probably not portable on PHP compiled without 64 bits support
    // double-check this when using this file on a new internet provider :-(
    if ($value > 0x7FFFFFFF) {
        $value = $value - 1073741824;
        $value = $value - 1073741824;
        $value = $value - 1073741824;
        $value = $value - 1073741824;
     }

    //var_dump($value);

    return $value;
}


// tests my_hash_string
function test_hash_string() {
   echo "."  ===>  ".my_hash_string("")."<br>";
   echo "t"."  ===>  ".my_hash_string("t")."<br>";
   echo "to"."  ===>  ".my_hash_string("to")."<br>";
   echo "tot"."  ===>  ".my_hash_string("tot")."<br>";
   echo "toto"."  ===>  ".my_hash_string("toto")."<br>";
   echo "toto "."  ===>  ".my_hash_string("toto ")."<br>";
   echo "toto e"."  ===>  ".my_hash_string("toto e")."<br>";
   echo "toto es"."  ===>  ".my_hash_string("toto es")."<br>";


   // note : the expected output is the following,
   //        any difference means there is a problem
   //        with the handling of 32 bits integers
   //        in the implementation of PHP we are using
   //   ===> 0
   // t ===> 1232605903
   // to ===> -29755991
   // tot ===> 360997608
   // toto ===> 1735262235
   // toto  ===> 145643908
   // toto e ===> -1371167412
   // toto es ===> 2070874248
}
```

6.3 Jenkins's hash function, in Java

```
// Hash function using Jenkins algorithm, see :
//    http://en.wikipedia.org/wiki/Hash_table
//    http://www.burtleburtle.net/bob/hash/doobs.html
//
// We will do our arithmetics in 64-bits signed integers,
// because there are no 32-bits unsigned integers in Java.
//
private static int Jenkins_hash_string(String my_string) {
    long value = 0;
    int i;
    long string_length = my_string.length();

    for (i = 0; i < string_length ; i++) {

        value += (long)( my_string.charAt(i));
        value = (value & 0xffffffffL);

        value += (value << 10);
        value = (value & 0xffffffffL);

        value ^= (value >> 6);
        value = (value & 0xffffffffL);
    }

    value += (value << 3);
    value = (value & 0xffffffffL);

    value ^= (value >> 11);
    value = (value & 0xffffffffL);

    value += (value << 15);
    value = (value & 0xffffffffL);

    // convert to 32 signed integer
    if (value > 0xffffffffL) {
        value = value - 1073741824;
        value = value - 1073741824;
        value = value - 1073741824;
        value = value - 1073741824;
    }

    return (int)value;
}
```