Stechec2

Pierre Bourdon
Nicolas Hureau

Jun 03, 2018

Contents

This documentation is still a work in progress. Sorry for empty sections: contributions are welcome!

Pages:

CHAPTER 1

Introduction

Stechec2 is client-server turn-based strategy game match maker used during the French national computing contest: Prologin.

Each year, we create a new game for the final and candidates are asked to develop an AI in 36 hours for this specific game. The final ranking will be determined at the end with a tournament where each AIs compete against each other.

Hence, stechec2 must be very flexible to accommodate with different games, and compatible with multiple programming languages such as: C, C++, Python, Java, Ocaml, Haskell, Rust, etc. We also need to isolate matches to share resources amongst untrusted code execution, and distribute them evenly over our own infrastructure.

CHAPTER 2

Usage

2.1 Build procedure

Stechec2's build system is based on Waf. The build procedure is composed of the two traditional steps: configure and build itself.

The configure step is the one in which you specify where you want to install Stechec2, with what games, how, etc. At the root of the source directory, run:

```
./waf.py configure --prefix=/home/user/stechec-install --with-games=tictactoe
```

This will configure the build system to build Stechec2 with the `tictactoe` game and to install it in `/home/user/stechec-install` (i.e. put executables in the `bin` subdirectory, libraries in `lib`, etc.). Note that this also creates the `build` directory, where all compilation artifacts (object files, test programs) will go.

Now, build Stechec2 and the corresponding games:

```
./waf.py build
```

And if everything went well, install it!

```
./waf.py install
```

2.2 Create your AI

Once the game is installed, you need to create an AI for it. To do so, generate the player environment (with different folders for each supported languages):

```
stechec2-generator player tictactoe player_env
```

You might need to install languages dependencies.

Go to the folder corresponding to the programming language you want to code in, and start editing the file `prologin`. A Makefile is provided to create a tarball containing all your source files (don't forget to update the Makefile if you add new files):

```
make tar
```

2.3 Launch a match

Stechec2 is based on a server-client architecture, hence you need to launch `stechec2-server` and one `stechec2-client` per player. This can be easily done with a wrapper called `stechec2-run` which runs everything needed. You only have to write a small YAML configuration file.

An example might be `config.yml`:

```
rules: /usr/lib/libtictactoe.so
map: ./simple.map
verbose: 3
clients:
  - ./champion.so
  - /path/to/other/champion.so
names:
  - Player 1
  - Player 2
```

A match can now be simply started with:

```
stechec2-run config.yml
```

2.4 Add spectators

To watch a game, you can add a spectator, which is a player that don't take part in the game.

First, compile your spectator:

```
cd /path/to/prologin2014/gui
make
```

Then, add those lines to your `config.yml`:

```
spectators:
  - /path/to/prologin2014/gui/gui.so
```

2.5 Debugging

To use gdb with stechec2 you can add the `--debug` option (or simply `-d`) followed by the client id you want to debug:

```
stechec2-run -d 1 config.yml
```

You can find other useful options by running:

```
stechec2-run -h
```

CHAPTER 3

Development

If you intend to contribute to Stechec2 or if you want to write your own game, here are useful tricks to ease your task. As a general note: you may be interested in taking look at the help message (`./waf.py --help`) to discover commands and options.

Now most importantly, add the `--enable-debug` option to the configure command so that Stechec2 is built with debugging information. This will enable you to run Stechec2 under GDB or any other debugger.

3.1 Using Clang

If you prefer Clang over GCC (for error messages, for instance), you can configure Stechec2 the following way (assuming you properly installed Clang++):

```
CXX=clang++ ./waf.py configure --with-games=...
```

Then build the project as usual.

3.2 Code coverage

Code coverage is basically the answer to "what part of the code is really executed?". It is particularly useful in the context of a testsuite. When some code is not covered (i.e. never executed), two conclusions can be drawed:

- either your testsuite misses testcases;
- either you have code that is useless... and thus that uselessly complexifies your codebase.

In order to compute code coverage reports, you have to configure Stechec2 with the `--enable-gcov` option. Then build Stechec2 as usual, execute it somehow (for instance running the testsuite) and then generate the report with the `coverage` command:

```
./waf.py configure --with-games=... --enable-gcov
./waf.py build --check # Build and run the testsuite
./waf.py coverage
```

At this point, you can open the `build/gcov-report.html` file in your favorite browser and discover what parts of your code are not tested/useless!

Note that code coverage does not work very well when using another compiler than G++. There exists `llvm-cov`, but our report formatter, `gcovr` mysteriously crashes when attempting to use it. So please use G++ when you want to compute code coverage. :-)

3.3 Address sanitizer

GCC or LLVM's address sanitizer is as useful as Valgrind when programming with manual memory management (such as in C or C++) to detect various memory issues. Using this feature is very easy in Stechec2: just use the `--enable-asan` configure option. ASAN will output messages on Stechec2's standard error output if it detects any issue. Note that when this happens in our testsuite, the corresponding testcases fail (which is good! such issues must be fixed!).

CHAPTER 4

Implementing games rules in stechec2

CHAPTER 5

Introduction

So you want to organize a big contest between IAs and you don't know where to start? This tutorial is for you! We're going to implement a basic Connect Four game in this tutorial, step by step, to understand the mechanics of stechec2's rules writing.

If you're stuck in some part, you can help yourself with the tictactoe game that has already been implemented in `games/tictactoe`.

CHAPTER 6

Think about The Game

We must define first *exactly* the rules of our game: how it works, who plays first, how many players a play can handle, who wins... The rules of the Connect Four are very simple: two players, a turn by turn game, only one action (drop a disk somewhere).

We can begin to describe the rules in the .yml:

```
$ mkdir connect4
$ cd connect4
$ vim connect4.yml
```

CHAPTER 7

Write the YAML

First, we must write some config boilerplate at the top of the file:

```
name: connect4  # The name of the game
rules_type: turnbased # The type of rules to follow

constant:
  # Place your constants here
enum:
  # Place your enums here
struct:
  # Place your structs here
function:
  # Place the functions of the API here
user_function:
  # Place the functions that the player sould implement here
```

7.1 The type system

Stechec2 uses its generators to implements some basic types in every supported languages. When a field requires a type in the yaml, you can use the built-in types (`int`, `bool` and `string`), the structs you defined, the enums you defined and `_ array` (where _ is a type itself). You can even use arrays of arrays (`int array array array`, for instance, will create a 3D matrix of ints).

7.2 The user functions

Depending of how the game works, the player should implement some functions in which he calls the actions of the API. Usually, this is done by:

- A function `init_game`, called at the start of the game, in which the player can create and initialize his objects
- A function `play_turn`, called at each turn, in which he can play (call the API functions).

- A function `end_game`, called at the end of the game, in which he can delete and free his objects.

We're going to use these functions for our connect4, which does not require more, but the player can have more functions to implement in a phase based game: `play_move_phase` and `play_attack_phase`, for instance.

In the yaml, a function is described like this:

- `fct_name`: the name of the function.
- `fct_summary`: a short documentation of the function.
- `fct_ret_type`: the return type of the function (bool, void, int array...).
- `fct_action` (yes/no): if the function is a game action (default value is no).
- `fct_arg`: the list of arguments that the function takes. Each item is a list containing:
  - The name of the argument.
  - The type of the argument.
  - A short description of its use.

So, let's write our user functions:

```
user_function:
  -
    fct_name: init_game
    fct_summary: Function called at the start of the game
    fct_ret_type: void
    fct_arg: []
  -
    fct_name: play_turn
    fct_summary: Function called at each turn
    fct_ret_type: void
    fct_arg: []
  -
    fct_name: end_game
    fct_summary: Function called at the end of the game
    fct_ret_type: void
    fct_arg: []
```

7.3 The constants

A constant is described by three fields:

- `cst_name`: the name of the constant.
- `cst_val`: the value of the constant.
- `cst_comment`: a short description of the constant.

The only constants we'll use in our game are the constants describing the size of the board, and the limit of players:

```
constant:
  -
    cst_name: NB_COLS
    cst_val: 8
    cst_comment: number of columns in the board
  -
    cst_name: NB_ROWS
```

(continues on next page)

(continued from previous page)

```
    cst_val: 6
    cst_comment: number of rows in the board
  -
    cst_name: NB_PLAYERS
    cst_val: 2
    cst_comment: number of players during the game
```

7.4 The enumerations

An enumeration is described by three fields:

- `enum_name`: the name of the enum.
- `enum_summary`: a short documentation of the enum.
- `enum_field`: a list of the different fields of the enum. Each field is a list containing:
  - The name of the field.
  - A short description of the field.

Our game will use only one enum, `error`, the return value of action functions.

We can write it quickly:

```
enum:
  -
    enum_name: error
    enum_summary:
      "Enumeration containing all possible error types that can be returned
      by action functions"
    enum_field:
      - [ok, "no error occurred"]
      - [out_of_bounds, "provided position is out of bounds"]
      - [full, "the selected column is full"]
      - [already_played, "you already played this turn, you cheater!"]
```

7.5 The structures

A structure is described by four fields:

- `str_name`: the name of the structure.
- `str_summary`: a short documentation of the structure
- `str_tuple` (yes/no): if set to "yes", in the languages that support it (Python and OCaml for instance), the structure will be represented as a tuple instead.
- `str_field`: a list of the different fields of the struct. Each field is a list containing:
  - The name of the field.
  - The type of the field.
  - A short description of the field.

The only struct we'll need is a `position { int x; int y; }`, to describe a position in the board:

```
struct:
  -
    str_name: position
    str_summary: Represents a position in the board
    str_tuple: yes
    str_field:
      - [x, int, "X coordinate (number of the column)"]
      - [y, int, "Y coordinate (number of the row)"]
```

7.6 The functions

The next part is to write the API that the player will use to play. The functions are usually separated in three kinds:

- The observers: functions that the player can call to see the state of a game. They can take some parameters to describe the information that the player wants, and they return the desired information.
- The actions: functions that the player can call to perform some action. They usually take some parameters to describe how the action should be executed and return an error. Errors are generally represented by an enum you have to implement. Note that you need to add a `fct_action: yes` field to the function.
- The state modifiers: functions that can cancel some actions or modify the state of the game.

So, here are the observers we'll implement:

- `my_player`: returns the ID of the current player
- `get_column`: returns the column (a int array corresponding to the disks of a column and their owners (-1 for "free", the id of the player else). The indice `0` of a column will represent its bottom.
- `cell`: returns the owner of the specified cell (-1 for "free").

The actions:

- `drop`: drop a disk at the specified column.

The modifiers:

- `cancel`: cancel the last action.

Add this at the end:

```
function:
  -
    fct_name: drop
    fct_summary: Drop a disk at the given position
    fct_ret_type: error
    fct_action: yes
    fct_arg:
      - [column, int, column where to drop a disk]
  -
    fct_name: my_player
    fct_summary: Return your player number
    fct_ret_type: int
    fct_arg: []
  -
    fct_name: get_column
    fct_summary: Return the column; indice 0 represents the bottom
    fct_ret_type: int array
    fct_arg:
```

(continues on next page)

(continued from previous page)

```
    - [number, int, number of the column]
-
  fct_name: cell
  fct_summary: Return the player of a cell (-1 for "free")
  fct_ret_type: int
  fct_arg:
    - [pos, position, position of the cell]
-
  fct_name: cancel
  fct_summary: Cancel the last played action
  fct_ret_type: bool
  fct_arg: []
```

And we're done!

CHAPTER 8

Generate the skeleton

Stechec2 provides a script to generate a skeleton of the rules. It really saves a lot of time, so don't skip this part!

If you have properly installed stechec2, you should have the generator in your PATH:

```
$ stechec2-generator -h   # Display a lot of useful help
$ stechec2-generator rules ./connect4.yml gen
$ mv gen/connect4/rules src
$ rm -rf gen
$ ls src
action_drop.cc  actions.hh  api.hh       entry.cc       game_state.hh rules.cc
action_drop.hh  api.cc      constant.hh  game_state.cc  interface.cc  rules.hh
```

You don't have to modify `constant.hh`, `entry.hh` and `interface.hh`. They are generated files that shouldn't be manually edited.

CHAPTER 9

The wscript

Stechec2 uses the waf.py Makefile-like to build the games. We need to create a `wscript` file in the root folder of our game, containing this:

```
#! /usr/bin/env python

def options(opt):
    pass

def configure(cfg):
    pass

def build(bld):
    bld.shlib(
        source = '''
            src/action_drop.cc
            src/api.cc
            src/entry.cc
            src/game_state.cc
            src/interface.cc
            src/rules.cc
        ''',
        defines = ['MODULE_COLOR=ANSI_COL_BROWN', 'MODULE_NAME="rules"'],
        target = 'connect4',
        use = ['stechec2'],
    )

    bld.install_files('${PREFIX}/share/stechec2/connect4', [
        'connect4.yml',
    ])
```

You can add source files to the `source` string. You don't need to change the rest for now.

CHAPTER 10

The rules

10.1 The loops

The first thing is to take a look at `rules.cc` and `rules.hh`. There are the three functions every rules should implement: `client_loop`, `spectator_loop` and `server_loop`. Writing these loops are painful: you have to handle the turns, the phases, the order of each players... luckily stechec2 provides some generic loops for some kind of games: `TurnBasedRules` and `SynchronousRules`. By adding the `rules_type` attribute in your configuration file, we don't need to worry about those functions.

If you're interested in how the generic loops work behind the scene, you can take a look at `stechec2/src/lib/rules/rules.cc`.

10.2 The game-state

We need to have a gamestate class which will contain the state of the game, and which we can interact with (the methods of this class will change the state of the game.) The majority of this part will be left as an exercise for the reader.

The basics of the GameState class are generated in the files `game_state.cc` and `game_state.hh`. Besides the already presents method, you'll also need for this game to define the following: `get_current_turn` and `increment_turn` which will do the needful with an internal counter, a `get_board` method which will return the 2D board, a `drop` to drop a disk somewhere, a `is_full` to check if one can play in a specific column, and finally, a `winner` method which will return the winner if there's one, -1 else.

Here's a template of the additional functions you'll need to implement:

```
void increment_turn();
int get_current_turn() const;
bool is_full(int column) const;
std::array<std::array<int, NB_COLS>, NB_ROWS> get_board() const;
int winner() const;
```

(continues on next page)

(continued from previous page)

```
void drop(int column, int player);
```

You will need to include `"constant.hh"` to make use of the constants.

10.3 Testing

Making unit test bit by bit as your rules are becoming more and more complex is really important: you don't want to test all the possible cases with custom champions.

Let's create a `src/tests` folder, where we'll put all our test files. The tests use googletest, you can find a reference documentation.

Here, we're going to create a `test-gamestate.cc` to test that the functions we just created are working well.

Here's a template for `test-gamestate.cc`:

```
#include <gtest/gtest.h>
#include "../game_state.hh"

class GameStateTest : public ::testing::Test
{
    protected:
        virtual void SetUp()
        {
            // Some code that will be executed before each test

            // Create an array of two players
            rules::Players_sptr players(new rules::Players {
                    std::vector<rules::Player_sptr> {
                        rules::Player_sptr(new rules::Player(0, 0)),
                        rules::Player_sptr(new rules::Player(1, 0)),
                    }
                }
            );

            gamestate_ = new GameState(players);
        }

        GameState* gamestate_;
};

TEST_F(GameStateTest, TestName)
{
    // Test content
}
```

You can then create as many tests as you want, for instance:

```
TEST_F(GameStateTest, CheckDropOverflow)
{
    for (int i = 0; i < NB_ROWS; i++)
    {
        ASSERT_EQ(gamestate_->is_full(0), false);
        gamestate_->drop(0, 0);
```

(continues on next page)

(continued from previous page)

```
    }
    ASSERT_EQ(gamestate_->is_full(0), true);
}
```

Create the following tests:

- *CheckFull*: checks that `is_full` returns `true` when the column is full
- **CheckDrop**: checks that the board obtained by dropping disks is valid
- **CheckWinner**: checks that you winner() function works correctly

To take tests into account, you first need to update your `wscript`

```
#! /usr/bin/env python

import glob
import os.path


def options(opt):
    pass

def configure(cfg):
    pass

def build(bld):
    bld.shlib(
        source = '''
            src/action_drop.cc
            src/api.cc
            src/entry.cc
            src/game_state.cc
            src/interface.cc
            src/rules.cc
        ''',
        defines = ['MODULE_COLOR=ANSI_COL_BROWN', 'MODULE_NAME="rules"'],
        target = 'connect4',
        use = ['stechec2'],
    )


    abs_pattern = os.path.join(bld.path.abspath(), 'src/tests/test-*.cc')
    for test_src in glob.glob(abs_pattern):

        test_name = os.path.split(test_src)[-1]
        test_name = test_name[5:-3]

        # Waf requires a relative path for the source
        src_relpath = os.path.relpath(test_src, bld.path.abspath())

        bld.program(
            features = 'gtest',
            source = src_relpath,
            target = 'connect4-test-{}'.format(test_name),
            use = ['connect4', 'stechec2-utils'],
            includes = ['.'],
            defines = ['MODULE_COLOR=ANSI_COL_PURPLE',
```

(continues on next page)

(continued from previous page)

```
            'MODULE_NAME="connect4"'],
        )

    bld.install_files('${PREFIX}/share/stechec2/connect4', [
        'connect4.yml',
    ])
```

To run the tests, you just have to build using the `--check` option:

```
./waf.py build --check
```

Running the testsuite is particularly useful when used along with coverage reports (see the *Development* section).

10.4 The actions

The actions are the only objects sent on the network. Let me expand on that part a bit. When you run a stechec2 match, you have a server and two clients. They load the same shared library that defines the rules of the game, and they create a local GameState (actually a linked list of gamestates, to allow a cancel() action that undoes actions). When a player wants to perform an action, the rules first check if the action can be made considering the current state of the game. If everything is okay, the stechec2 client "apply" the action to the gamestate and send the action over the network. The server then receives the action, and checks if it can be made too. If not, there's a big synchronisation problem (or possibly an attack), so the server disconnects the client. Else, the server applies the action locally to his gamestate and broadcast the action to the other players (so that they can do the same with their gamestates).

An action must define five functions that will be used by the rules:

- **check(gamestate)**: checks that the action can be applied on the gamestate ;
- **apply_on(gamestate)**: applies the action to the given gamestate ;
- **handle_buffer(buffer)**: used to serialize the action object to a buffer ;
- **id()**: returns the ID of the action (usually an element of an enum) ;
- **player_id()**: returns the ID of the player that sent the action ;

Most of these functions are already implemented automatically in `actions.hh`, but we still need to code the `check` and `apply_on` functions. Note that `check` should return an element of the error enumeration we've defined in the rules (see `constant.hh`): { OK, OUT_OF_BOUNDS, FULL, ALREADY_PLAYED }.

10.5 The API

In the bunch of files you've previously generated, there is a file called `api.cc` that will describe what happens when the player calls a function during the game. These functions are directly "translated" in the language from which they are calling them, so you just have to implement the behaviour as if everyone played in C++.

The observers are a realy easy part, you just have to return some values from the GameState and the rules::Player objects. For instance with my_player:

```
int Api::my_player()
{
    return player_->id;
}
```

Implement all the other observers: `get_column` and `get_cell`. In order to call our gamestate-specific functions, you need to use the `game_state_` member.

The `cancel` function is already implemented in stechec2. To call it you just have to do this:

```
bool Api::cancel()
{
    if (!game_state_->can_cancel())
        return false;
    actions_.cancel();
    game_state_ = rules::cancel(game_state_);
    return true;
}
```

Internally, there's a linked list of gamestates. The `rules::cancel` function simply removes the current gamestate and returns the last.

The actions in the API are already implemented. Each action, first calls the appropriate `check` function, and if this returns OK, calls `apply_on` to update our gamestate, and add the action to the actions list.

10.6 The rules object

Let's typedef the function that will be called by the player as void*()'s in our rules.hh:

```
typedef void (*f_champion_init_game)();
typedef void (*f_champion_play_turn)();
typedef void (*f_champion_end_game)();
```

Then add these attributes to the Rules class:

```
protected:
    f_champion_init_game champion_init_game;
    f_champion_play_turn champion_play_turn;
    f_champion_end_game champion_end_game;
```

In the `Rules` constructor, we have to retrieve the champion library

```
Rules::Rules(const rules::Options opt)
    : TurnBasedRules(opt), sandbox_(opt.time)
{
    if (!opt.champion_lib.empty())
    {
        champion_dll_ = std::make_unique<utils::DLL>(opt.champion_lib);

      champion_init_game_ =
          champion_dll_->get<f_champion_init_game>("init_game");
      champion_play_turn_ =
          champion_dll_->get<f_champion_play_turn>("play_turn");
      champion_end_game_ =
          champion_dll_->get<f_champion_end_game>("end_game");
    }

    GameState* game_state = new GameState(opt.players);
    api_ = std::make_unique<Api>(game_state, opt.player);
    register_actions();
}
```

Then we can overload the functions defined in <rules/rules.hh> to satisfy our needs. For instance, we want to overload at_player_start, player_turn and at_player_end to execute the init_game, play_turn and end_game client functions. To do so, we'll use the sandbox object:

```
void Rules::at_player_start(rules::ClientMessenger_sptr)
{
    try
    {
        sandbox_.execute(champion_init_game_);
    }
    catch (utils::SandboxTimeout)
    {
        FATAL("player_start: timeout");
    }
}

void Rules::player_turn()
{
    try
    {
        sandbox_.execute(champion_play_turn_);
    }
    catch (utils::SandboxTimeout)
    {
        FATAL("player_turn: timeout");
    }
}

void Rules::at_player_end(rules::ClientMessenger_sptr)
{
    try
    {
        sandbox_.execute(champion_end_game_);
    }
    catch (utils::SandboxTimeout)
    {
        FATAL("player_end: timeout");
    }
}
```

We also need to implement functions such as start_of_player_turn, end_of_player_turn, and is_finished that will call our gamestates functions:

```
void Rules::start_of_player_turn(unsigned int /* player_id */)
{
    api_->game_state()->increment_turn();
}

void Rules::end_of_player_turn(unsigned int /* player_id */)
{
    // Clear the list of game states at the end of each turn (half-round)
    // We need the linked list of game states only for undo and history,
    // therefore old states are not needed anymore after the turn ends.
    api_->game_state()->clear_old_version();
}

bool Rules::is_finished()
{
```

(continues on next page)

(continued from previous page)

```
    const GameState* st = api_->game_state();
    return st->winner() != -1;
}
```

In stechec2, there is a difference between turns and round. A **round** is made up of 2 **turns**, one for each player. You can therefore overload the same functions but for round specific needs, such as `start_of_round`, `end_of_round`, etc.

And that's it!

CHAPTER 11

FAQ

Server-side timeout **must** be greater than sandbox timeout, otherwise the gamestate might be inconsistent.