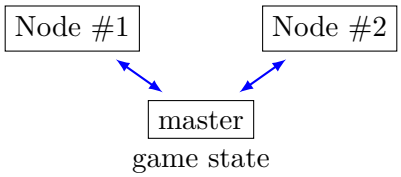
Assignment P4

1 Introduction

In the fourth assignment, we implement a distributed computer game, which simulates a general control problem that may arise, e.g., in industrial scenarios.

The game will communicate over the Lownet protocol we are familiar with by now. The new aspect is the strict real-time requirements for control actions.



1.1 High-level functionalities

What is provided: The starting point for this assignment includes again the skeleton code that implements many functionalities, but few critical things are missing!

1. The new element is the real-time game protocol, which includes packet formats for (i) registration, (ii) status updates and (iii) game actions.
2. Important definitions can be found from `games.h` file.

Your tasks:

The tasks have been divided into two milestones. The first milestone is a soft target for the first week. The second milestone corresponds to fully functioning submission.

Milestone I:

1. Task 1: Add checksums to action packets.
2. Task 2: Implement the node-specific encryption for game protocol packets.

The details are given in the next section.

After Milestone I, the focus shifts on developing robust control actions (game actions) under the hard-deadline constraints.

Milestone II:

1. Task: Implement better control actions.

   Your solutions is evaluted with respect to the following three criterion:

   (a) Actions must be taken in time!

   (b) Actions must be feasible

   (c) It must be a winning strategy against the random player:

   - Objective: win 8 games in a row

2 Specification for Communication Protocol

A high level description of the new game protocol is given in the slides. The communication protocol details are not your main concern, but it is still helpful to understand what happens behind the scenes when implementing the necessary functionalities for the game itself. For example, network may introduce delays and hence you should probably communicate your action before the last millisecond!

2.1 Operation

The games are played in following three stages:

1. **Registration:** nodes first register to the master node in order to join a game. Master notes down the relevant information and pairs each node to another node.
2. **Status updates and Actions:** A game starts with a status update that contains an empty board. The player 1 then makes its first move by sending an appropriate **action packet**.
   If the move is legal (in time, etc.), the master node approves it and sends a new round of status updates. Then it is player 2' turn. This repeats until the game ends.
3. **Game over:** When the game ends, the master node sends a final status update and announces the winner, or a tie result.

3 Tic-Tac-Toe Game

The game played on the 30×30 board, which squares are numbered as depicted in Figure 1. The player 1 starts by drawing a cross to one empty square. Then it is player 2's turn, who draws a null (zero). This is repeated until (i) either player has 5 squares in a row (horizontal, vertical, or diagonal), or (ii) a predetermined maximum number of rounds is reached. In the latter case, the master announces a tie.

It is also possible that some markers vanish. This will happen rarely, or never. Hence, the node is suppose to update its state information before making a decision.

(it is still possible and allowed to preplan moves while the opponent thinks, while taking the above exception into the account).

4 Milestone I

4.1 Task 1: checksum

First step is to familiarize yourself with the sparse and dense (packet) representations of the game state. The sparse encoding reserves one byte in memory for each square, using the ordering depicted in Figure 1.

The dense encoding packs five squares into each byte, and hence the memory requirements decrease from 900 to 180 bytes. The encoding works using base-3,

$$y_i = x_{5i} + 3 \cdot x_{5i+1} + 3^2 \cdot x_{5i+2} + 3^3 \cdot x_{5i+3} + 3^4 \cdot x_{5i+4},$$

where $x_i \in \{0, 1, 2\}$ and $i = 0, \ldots, 179$. Hence, $0 \leq y_i < 243$ for all i.

Third potentially useful encoding uses base-2, where we use two bits for each square with the following convention:

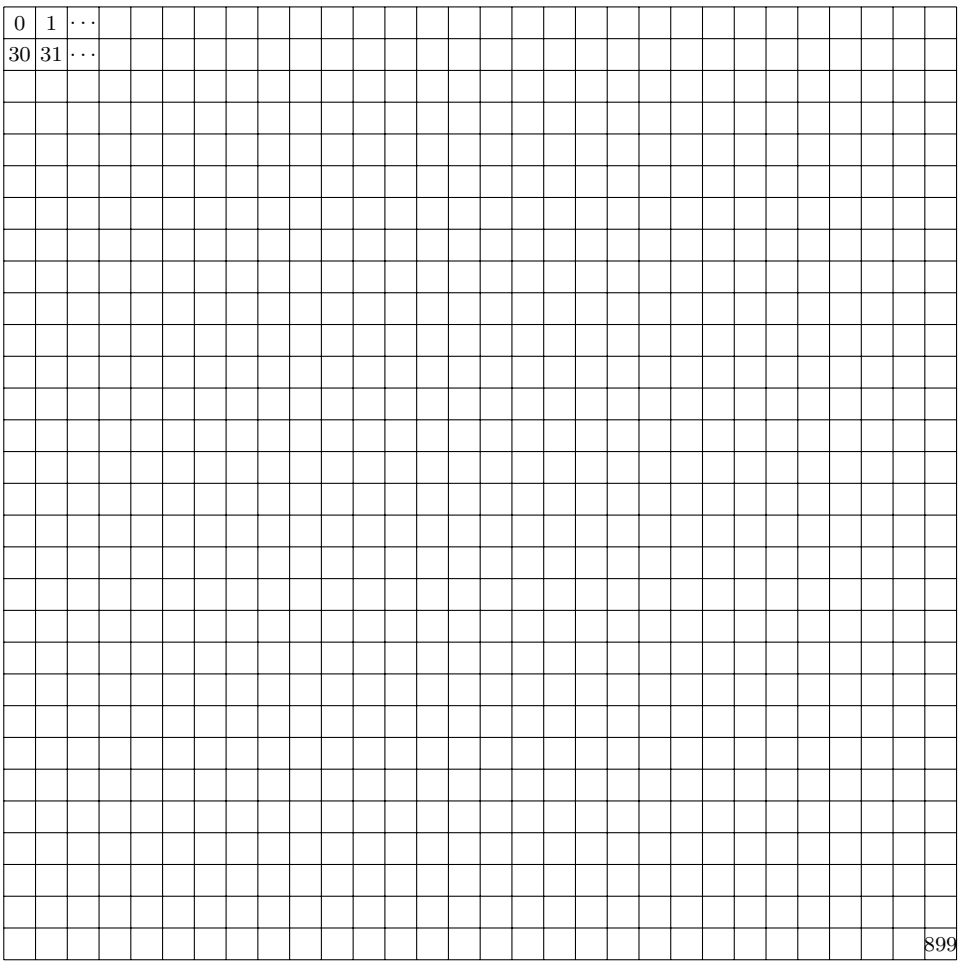


Figure 1: 30x30 board and the numbering of its squares.

00 empty
01 player 1
10 player 2
11 empty (2)

Hence, we have two possibilities to encode empty squares, 00 and 11. Effectively, we have 1 bit of information that we can convey per square!

We could use this one bit, e.g., to indicate the potential value of each square. For example, 00 could used for those squares whose distance from the marked squares is at least two squares in any direction (including diagonal). Then the squares whose distance is less would be marked with 11. Classifying the all squares of the board into the two classes is a good start for a heuristic strategy. **You are encourages to explore further in this direction!**

However, in the specification we use only the patterns, 00, 01 and 10. The two bit encoding encodes four squares into one byte, and thus each board requires $900/4 = 225$ bytes, which is slightly more than the maximum payload in the lownet frames.

Let B_k denote the board encoding with k squares per byte, where $k = \{1, 4, 5\}$. The checksum for actions will be based on two bit encoding! The node is suppose to return the CRC-24 code (3 bytes) calculated for the 225 byte long array of B_4 after the proposed action.

Example: *Suppose the squares* $0, \ldots, 4$ *have* x *in square* 1 *and* o *in square* 3*, and ALL the rest are empty. Hence,* B_1 *is* $\{0, 1, 0, 2, 0, 0, \ldots\}$*. The encoding with* B_4 *boils down to bit string* 00 01 00 10 00 00 ... *so that the first byte is*

$$1 \cdot 2^2 + 2 \cdot 2^6 = 4 + 128 = 132.$$

The same board when encoded with B_5 starts with $1 \cdot 3^1 + 2 \cdot 3^3 = 57$.

4.2 Task 2: Node-specific encryption

The goal is that each node uses its own AES key to encrypt all packets of the game protocol. The private key can be found from the "credentials" document, i.e., this is a shared secret between a node and the master node.

The main purpose of this is to prevent other nodes from interfering the game. Here we have a game, so the stakes are not so high, but given we are simulation a real-time control system, the stakes could be much higher!

Exact details on the frame format will be provided in a separate document. Master node will switch to this mode of operatation at a date that will be announced later. Until then, the game protocol shall use the same encryption key "0" as otherwise.

5 Milestone II

You shall work on the game policy or strategy that is specified in the function `my_policy`. This function is a task that reads game states from the (FreeRTOS) queue, solves them as puzzle, and returns the best possible action they can figure out in the given time budget.

The goal is to come up with a strategy that almost always wins the pure random strategy of the random player.

Communication is allowed via Lownet protocol, but not otherwise. That is, you shall not offload the game to your PC or elsewhere via the serial line (or WiFi).

Heuristic strategy of the random player

The random player is based on the following heuristic rules:

1. Identify all free squares that are adjacent to any existing marker.
2. Among these, choose a random square uniformly in random[1]

6 Testing Procedure

The test device at Gróska will engage your node in a sequence of tic-tac-toe games once you register to it with the `/game` command (as provided in the skeleton).

Tests: (evaluation) Your implementation will be tested as follows:

1. Device has to join a game at master node and take valid actions with the **checksum** correctly computed.
2. Device shall win eight (8) consecutive games against the random player. Note that the master MAY be your opponent, but that is irrelevant as all the communication goes via the master node anyway.

[1] To this end, the node uses a simple LCG generator, so the choice is not "very random" in practice.

Appendix A: Skeleton code

In principle, you only need to touch `tictac_node.c` file and implement the missing features there.

However, it is not forbidden to modify other files, especially if it leads to a more performant solution. So on your own risk!

```
#include <stdint.h>
#include "tictactoe.h"

/*
 * Identical to lownet_crc() - there is no need to touch this!
 */
uint32_t crc24( const uint8_t *buf, size_t len )
{
    uint32_t reg = 0x00777777;                  // shift register
    static const uint32_t poly = 0x1800463ul;  // G(x)

    void process_byte( uint8_t b )
    {
        for(int i=0; i<8; i++)
        {
            reg = (reg<<1) | (b&1);
            b   = b>>1;
            if ( reg & 0x1000000ul )
                reg = (reg^poly);      // take mod G(x)
        }
    }

    for(size_t i=0; i<len; i++)
        process_byte( buf[i] );
    return reg;
}

/*
 *  Milestone I: encode the board using 2 bits per square
 */
uint32_t tictac_checksum( const tictactoe_t *b )
{
    uint8_t b4[ TICTACTOE_N2 ];  // work here?

    /* TODO: encode the given board b into b4 */

    /* Then compute the CRC code */
    return crc24(b4, TICTACTOE_N2 );
}

/*
 *  Milestone II:
 *
 *  - Implement some super duper strategy for player s!
 *  - Time budget [2sec,10sec] is given in milliseconds
 *  - Return value 0 on making a succesful move (always!)
 */
int tictac_move( const tictactoe_t *b, int *xp, int *yp,
                 uint8_t s, uint32_t time_ms )
{
    /* TODO: figure out a better move than the random! */
    return tictac_auto( b, xp, yp, s );
}
```