Cédric Lood
Jean-Luc Zirani

17 décembre 2012

Table des matières

1 Introduction

The present report provides some documentation related to the second project in Real Time Operating Systems (Fall 2012). The purpose of the project was to develop a piece of software that uses the Message Passing Interface (MPI) library. The software developed is based on the game Cluedo, and allows for a human player to play against several AIs (2 to 5 instances). The project was developed with the goal of running it on Hydra (High performance computing cluster accessible to ULB students).

2 How to use the software ?

Although the software was designed to be run on the Hydra cluster, it should also work on any GNU/Linux computer, but if you plan on running it on your own computer, please consider installing the MPI libraries on your system beforehand.

To compile the software, please use the makefile located at the root of the project folder. Once connected to Hydra, the following commands will get you up and running :

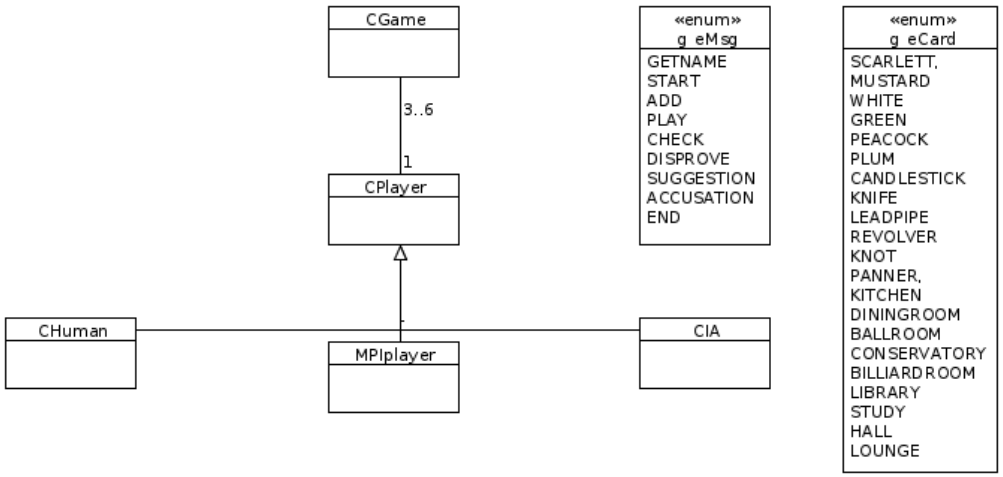
- module load openmpi/1.4.3/gcc/4.6.1
- make

This will produce an executable named Cleudo located in the *bin* folder. To run it, first move into the *bin* folder. Then consider the number of players you would like to have in the game, say 3 for the sake of example, and run the game with the following command : *mpirun -np 3 cleudo*

As you can see from the example, the number of players is passed to the executable using MPI's *-np* argument. Please note that since we need to interact with the user (using the input/output capabilities of the shell), we have not supplied a job preparation file. Once the executable is launched, the human player will be able to use the shell window in order to play.

3 Implementation overview

3.1 Class diagram



3.2 Game

The *Game* part of the code resides in the *CGame* class. It can be seen as a coordinator. That is, it does the following :

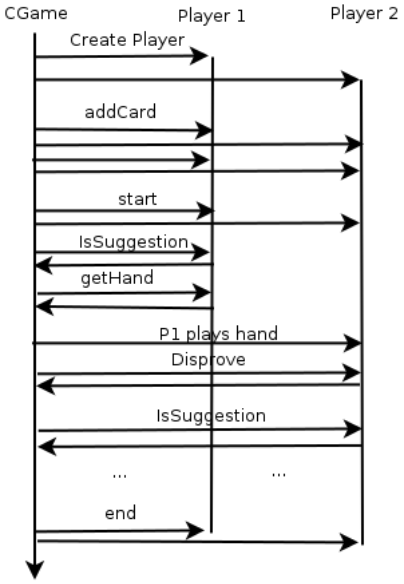
- it distributes the cards among the players and keeps 3 cards secret(a person, a weapon and a room).
- it coordinates the player and gives them turn
- it handles the end of the game

3.3 Player

We have defined the class *CPlayer* to be an interface with all the prototypes of functions. From this class we have then 3 specialized subclasses :

- CHuman : represents the human player
- CIA : represents the AI (we have kept the intelligence level very basic as the game was designed to teach us to use MPI)
- MPIplayer : wraps the prototype functions into MPI messages so we can add parallel processing in the software.

3.4 Activity diagram



The above activity diagram shows a typical instance of a game. As you can see, we first create the players. We then proceed to conceal 3 cards and distribute the rest of them among the players in a random fashion.

Once all the players have their cards we signal them that the game will be starting soon (actually, we synchronize the different processes). After that, we are ready to start taking turns between players, and ask the first player for its hand. He can then make a suggestion or an accusation, and we ask the next player to try and disprove at least one of the cards. Note that we also notify the other players the suggestion/accusation that was made.

If no one is able to disprove the card, the player (human or AI) knows that the card must be concealed. In the case of an accusation, we verify if he is correct. If he is, we

signal the end of the game. Otherwise he is disqualified and will only be authorized to disprove further suggestions, but not to make suggestions or accusations on his own.

Once a player has reached a successful accusation, we proceed to announce to the other players that he has won the game and the game ends.

3.5 MPI

The basic idea is to separate the implementation of the game per se and that of the parallelism architecture (MPI). In the MPI part, we have a root process and a child process. The root process handles the launch of the game and the human player. The child process is in charge of handling the MPI messaging system from the root process to the AI players.

As a design choice, we have decided to use only *send* and *recv* instead of *bcast* in order to make the code of the game less dependent on MPI.

4 Conclusion

During the conception of this software, we have grown quite fond of the MPI library. It is an efficient way to abstract the communication between related processes and its usage is quite natural after a short while.

MPI also offers an elegant abstraction of the messaging systems regardless of the fact that the communication relies on the network, on multiple cores, .. there also are some optimisations which are available based on this consideration (such as bcast, gather, scatter, barrier, ... - see design choice in the section on MPI).

A Coding Conventions

In order to be consistent in the naming of variables throughout the code, we have decided to use the following naming conventions :

Variable Names

The format used is : [*scoping*][*static*][*const*]_[*type*][*nameOfVariable*]

Scoping

- g : global
- m : class variables
- f : functions
- nothing : local to a scope
- static : s
- const : c

Primitives

- vec : vector
- str : std string and C-style string
- ui : unsigned int
- i : int
- OGS : Oppa Gangnam Style

Classes, Structures

- class : CName
- struct : SName
- enum : EName

Examples

- CPlayer : Class named "Player"
- m_vecPassive : Class attribute of type vector, named "Passive"
- m_uiMMin : Class attribute of type unsigned int named "MMin"