Prof. Dr. Thomas Schmidt
HAW Hamburg, Dept. Informatik,
Raum 480a, Tel.: 42875 - 8452
Email: t.schmidt@haw-hamburg.de
Web: http://inet.haw-hamburg.de/teaching

Verteilte Systeme

Aufgabe 1: Client/Server-Anwendung "Verteiltes Tic Tac Toe"

Ziele:

1. Java RMI kennen und verstehen lernen
2. Eine interoperable Client-Server-Anwendung erstellen
3. Fehlersemantiken anwenden und Fehlertoleranzen implementieren

Vorbemerkungen:

In dieser Aufgabe wird ein verteiltes Tic Tac Toe Spiel entwickelt, welches aus einem TTT-Client und einem TTT-Server besteht. Der TTT-Client kontaktiert den TTT-Server mit Anmeldedaten und zeigt das (3x3) Spielfeld. Der TTT-Server verwaltet die angemeldeten TTT-Clients, führt das (eine) Spiel durch und merkt sich den Zustand des Spiels.

Aufgabenstellung:

Entwickeln Sie ein Client/Server-Paar gemäß nachfolgenden Spezifikationen, welches auf beliebigen unterschiedlichen Laborrechnern über JAVA/RMI miteinander kommunizieren kann. Hierfür

1. konzipieren Sie zunächst den Programmaufbau, insbesondere die Verteilungskomponenten und ihre Fehlersemantiken;
2. implementieren Sie hiernach Client und Server unter strikter Einhaltung der Kommunikations-schnittstelle;
3. testen Sie dann Ihre Programme, insbesondere auf Interoperabilität mit Nachbargruppen;
4. untersuchen Sie die RMI-Kommunikation mit dem Netzwerk-Sniffer, dokumentieren Ihre Beobachtungen und begründeten Interpretationen in einem Protokoll.

Spezifikation:

Server:

1. Der TTT-Server nimmt über seine entfernte Methode `findGame()` die Spielregistrierung eines TTT-Clients entgegen, speichert die Registrierung und wartet auf den nächsten Client, um ein Spiel zwischen beiden zu erstellen.
2. Sobald sich ein zweiter TTT-Client registriert, wählt der TTT-Server einen Client als Start-spieler und gibt an beide Clients eine gemeinsame ID, den Namen des Gegners, sowie die Info wer anfängt zurück. Weitere Clients werden in einer Warteschlange gehalten.
3. Der TTT-Server nimmt die Spielzüge der Clients entgegen, aktualisiert seinen Spielstand, entscheidet, ob gewonnen wurde und gibt an beide Clients jeweils den Zug des Gegenspielers, ggfs. wer gewonnen hat oder einen Fehler gemäß Interface zurück.
4. Der Server implementiert eine ‚at-most-once' Fehlersemantik, d.h. er verarbeitet Spielzüge niemals mehrfach.
5. Der Server implementiert je Client einen Timer mit konfigurierbarem Timeout ***s***, der die höchstmögliche Zeit bis zum nächsten Spielzug vorgibt und beim Timeout den Client-Zustand und den Spielstand verwerfen lässt. Vor Ablauf dieses Timers erlaubt der Server den Clients, wieder in eine unterbrochene Verbindung zum laufenden Spiel einzutreten.
6. Der Server bleibt robust gegen Client-Fehlverhalten und protokolliert seine Aktionen in einem aussagekräftigen Logfile.
7. Seine entfernte Schnittstelle registriert der Server unter dem Namen " TicTacToeAService ".

Schnittstelle:

Client und Server kommunizieren über folgendes Remote Interface:

```
public interface TicTacToeAService extends Remote {
    // Returns a map with four keys:
    //     "Game ID", "Opponent Name", "First Move", "Move"
    // Each key maps to a string containing the respective value:
    // * Game ID and Opponent Name can be any reasonable strings.
    // * First Move is a string from:
    //     ["your_move", "opponent_move", "no_opponent_found"]
    // * Move is an empty string ("") if this player begins. Otherwise
    //   it is the move ("x,y") performed by the player that went first.
    //
    // Steps:
    // - if game is waiting for player:
    //   * assign player to game
    //   * randomly chose who begins
    //   * write info into state
    //   * if this player goes first:
    //     > return the tuple (see above)
    //   * if this player goes second:
    //     > signal the other player
    //     > wait for the other player to make a move (see `makeMove`)
    //     > return the tuple (see above)
    // - else:
    //   * create a new game in a "waiting-for-player" state
    //     > wait for other player to join
    //     > if timeout:
    //       - return [0, "", "no_opponent_found", ""]
    //     > if opponent joins:
    //       - read data from state
    //       - if this player goes first:
    //         * return the tuple (see above)
    //       - if this player goes second:
    //         * wait for the other player to make a move (see `makeMove`)
    //         * return the tuple (see above)
    public HashMap<String, String> findGame(String clientName)
                                          throws RemoteException;


    // Returns a String from ["game_does_not_exist", "invalid_move",
    //   "opponent_gone", "you_win: x,y", "you_lose: x,y", "x,y"]
    //
    // Grid: 0,0 is on the top left
    //
    // Steps:
    // - if game with `gameId` exists:
    //   * if move is invalid:
    //     > "invalid_move"
    //   * if move ends game:
    //     > note win
    //     > signal other player
    //     > return "you_win: x,y" | "you_lose: x,y"
    //   * else:
    //     > signal other player
    //     > wait for other player to make a move
    //     > if timeout:
    //       - return "opponent_gone"
    //     > if opponent makes a move:
    //       - if move ends the game:
    //         * return "you_win: x,y" | "you_lose: x,y"
```

```
    //       - else:
    //         * return "x,y"
    // - else:
    //   * return "game_does_not_exist"
    public String makeMove(int x, int y, String gameId)
                                          throws RemoteException;

    // Returns a list with all moves in the game with ID gameId.
    //
    // Each string has the pattern "name: x,y" or "winner: NAME".
    // A winner must be the last element in the array and follows
    // the move that won the game.
    public ArrayList<String> fullUpdate(String gameId)
                                          throws RemoteException;
}
```

Client:

1. Der TTT-Client Client nimmt beim Start eine Adresse und einen Port für den Verbindungsaufbau entgegen zusammen mit einem Namen für den Spieler.
2. Der Client meldet sich beim Server mit dem Namen an.
3. Nach der Nachricht zum Spielstart durch den Server führen die Clients abwechselnd Züge aus und aktualisieren ihr graphisches Spielfeld dem Spielverlauf entsprechend.
4. Verliert ein TTT-Client die Verbindung zum Server (oder stürzt ab), kann er sich mit denselben Anmeldeinformationen wieder in das laufende Spiel einbinden und den Spielstand aktualisieren (s. Interface).

Online-Dokumentation zu Java RMI:

- http://docs.oracle.com/javase/tutorial/rmi/overview.html
- http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

Achten Sie besonders auf die erforderlichen Sicherheitseinstellungen für die RMI Registry!