Fachbereich 3 Mathematik und Informatik
Arbeitsgruppe Datenbanksysteme
M. Gogolla, R. Kollmann, A. Lattner, S. Prelle,
M. Richters, H. Shi, A. Werner

9. April 2001

Softwarepraktikum 2001
Scotland Yard Protokollbeschreibung

Für den Datenaustausch zwischen Client und Server bei der Scotland Yard-Aufgabe ist ein TCP-basiertes ASCII-Protokoll vorgesehen. Daß bedeutet im Klartext, daß eure Programme in der Lage sein müssen, menschenlesbare Kommandos zu erzeugen und zu verstehen und daß ihr zum Testen das Telnet-Kommando verwenden könnt.

Das zu verwendende Protokoll ist vorgegeben und soll es ermöglichen, daß ihr eure Anwendungen gruppenübergreifend testen könnt. Dieses Dokument enthält die Protokollspezifikation.

Kommando-Aufbau

Kommandos, die an den Server gesendet werden, bestehen aus jeweils einer Zeile und haben folgenden Aufbau:

```
<commandname> [args...]
```

Bei der Beschreibung der Kommandos sind optionale Argumente in eckigen Klammern aufgeführt. Als Trennzeichen zwischen den einzelnen Elementen eines Kommandos werden Leerzeichen und Tabulatoren verwendet.
Kommandos, die vom Server an die Clients gesendet werden, gibt es in zwei Varianten: einzeilig oder mehrzeilig. Einzeilige Kommando sehen wie folgt aus:

```
syserv: <commandname> [args...] comment
```

`comment` meint dabei einen Kommentar, der hinter dem letzten Argument beginnt und bis zum Ende der Zeile geht. Kommentare lassen sich quasi nur dadurch von Argumenten unterscheiden, daß bekannt ist, wieviele Argumente zu einem Kommando gehören.

Mehrzeilige Kommandos fangen identisch wie einzeilige Kommandos an und markieren ihre letzte Zeile dadurch, daß diese nur einen Punkt enhält.

```
        syserv: <commandname> [args...] comment
<data-1>
<data-2>
...
<data-n>
.
```

Synchrone Kommandos

Nach jedem ausgeführten Kommando schickt der Server eine Antwort an den Client zurück. Welche Antwort bei der erfolgreichen Ausführung eines Kommandos erwartet wird, ist bei den einzelnen Kommandos angegeben. Im Fehlerfall antwortet der Server mit der Nachricht

```
syserv: error <errno>
```

wobei `<errno>` ein Dezimalzahl ist, die die Fehlerursache beschreibt.

Der Server soll mindestens die folgenden Kommandos verstehen, und Clients sollten sich nur auf diese Kommandos verlassen.

1. `register <name>`

   Um an einer Partie teilzunehmen oder anderen Benutzern Nachrichten zu schicken, muß man sich zunächst mit diesem Kommando mit einem Namen registrieren.

   ```
   syserv: ok
   ```

2. `help [<command>]`

   Ohne Argumente wird eine Liste aller Kommandos angezeigt. Wird als Argument ein Kommandoname übergeben, wird Hilfe zu diesem Kommando gegeben.

   ```
   syserv: help
   Hilfe-Text
   ...
   Hilfe-Text
   .
   ```

3. `exit`

   Mit diesem Kommando wird die Verbindung zum Server abgebrochen (und damit natürlich auch alle Spiele abgebrochen).

   ```
   syserv: ok
   ```

4. `who [addr|host]`

   Liste aller registrierten Benutzer - eine Zeile für jeden Benutzer - anzeigen. Jede Zeile enhält den Namen des Benutzers und evtl. die IP-Adresse bzw. den hostname, von dem der Benutzer die Verbindung aufgebaut hat, wenn die Option `addr` oder `host` angegeben wurde. Die Implementierung der Optionen `addr` und `host` ist freiwillig.

   ```
   syserv: users
   user-1
   user-2
   ...
   user-n
   .
   ```

5. `send <user-list> <message>`

   Nachrichten können an einen oder mehrere, durch Kommata getrennte Benutzer geschickt werden. Als Adressaten können die Namen registrierter Benutzer oder die Namen von Spielen verwendet werden. Wird der Name eines Spiels angegeben, wird die Nachricht an alle beteiligten Spieler und Beobachter des Spiels geschickt.

```
syserv: ok
```

6. `create <game> <role> <grmodel>`

   Neue Spiele werden mit diesem Kommando unter einem neuen Namen und mit dem angegebenen game resource model eröffnet. `<role>` gibt an, in welche Rolle der Spieler übernehmen möchte. Dabei bedeutet:

   0: Mr.X
   1: erster Detektiv
   2: zweiter Detektiv

   usw.

   Dabei ist zu berücksichtigen, wieviele Detektive laut dem gewählten game resource model an dem Spiel teilnehmen können. In der Antwort wird dem Spieler seine Startposition mitgeteilt.

   ```
   syserv: joined <position>
   ```

7. `join <game> <role>`

   Mit diesem Kommando kann ein Benutzer an einem Spiel in der angegeben Rolle teilnehmen. In der Antwort wird dem Spieler seine Startposition mitgeteilt.

   ```
   syserv: joined <position>
   ```

8. `move <game> <role> <to> <ticket>`

   Spielzüge werden mit diesem Kommando ausgeführt.

   ```
   syserv: ok
   ```

9. `resign <game>`

   Ein laufendes Spiel kann verlassen werden.

   ```
   syserv: ok
   ```

10. `watch <game> [on|off]`

    Die Beobachtung von Spielen kan ein- und ausgeschaltet werden. `watch` ohne Argumente ist synonym zu `watch on` .

    ```
    syserv: ok
    ```

11. `status <game>`

    Zeigt den Status eines Spiels an.

    ```
    syserv: status <game> <grmodel> <map> <role>
    user-1 *        tickets...
    user-2 position tickets...
    ...
    user-n position tickets...
    .
    ```

Jeder der durch das game resource model festgelegten Spieler wird durch eine Zeile beschrieben, die seinen Namen bzw. 'Missing', seine Position und die Anzahl seiner Tickets angibt. `tickets` ist eine durch Leerzeichen getrennte Liste von so vielen natürlichen Zahlen wie es verschiedene Ticket im verwendeteten game resource model gibt.
Anstelle der Position von Mr.X, die geheimgehalten werden muß, wird ein `*` übergeben, außer an Mr.X selbst. Ruft Mr.X dieses Kommando auf, steht an dieser Stelle seine momentane Position.
Wenn dieses Kommando von einem Teilnehmer des Spiels aufgerufen wird, enthält `role` die Nummer seiner Rolle, ansonsten -1.

12. `list grmodels [<map>]`

    Zeigt eine Liste aller game resource models oder nur derjenigen für eine bestimmte Stadt `<map>` . Diese Liste kann evtl. auch leer sein, wenn keine passenden models vorhanden sind.

    ```
    syserv: grmodels
    grmodel-1
    grmodel-2
    ...
    grmodel-n
    .
    ```

13. `list maps`

    Zeigt die Liste der vorhanden Stadtpläne an.

    ```
    syserv: maps
    map-1
    map-2
    ...
    map-n
    .
    ```

14. `list games`

    Zeigt die Namen der eröffneten Spiele an.

    ```
    syserv: games
    game-1 grm-1 map-1 0|1
    game-2 grm-2 map-2 0|1
    ...
    game-n grm-n map-n 0|1
    .
    ```

    Jedes Spiel wird durch eine Zeile beschrieben, die Name, game resource model und die zugehörige map enthält. Das letzte Argument ist entweder '0', wenn sich nicht genügend Spieler (mit `join`) für das Spiel angemeldet haben, oder '1', wenn das Spiel vollständig ist und damit schon begonnen hat.

15. `get <map>`

    ```
    syserv: map <map>
    map-text
    ...
    map-text
    .
    ```

16. `get <grmodel>`

```
syserv: grmodel <grmodel>
grmodel-text
...
grmodel-text
.
```

Asynchrone Kommandos

Die Clients müssen auch auf Nachrichten des Servers reagieren können, die nicht unmittelbar als Antwort auf ein Kommando, also asynchron, geschickt werden.

Dies sind die folgenden Nachrichten:

- `syserv: hello`

  Diese Nachricht wird einmal sofort nach dem Aufbau der Verbindung vom Server zum Client geschickt.

- ```
  syserv: start <game>
  user-1 *        tickets...
  user-2 position tickets...
  ...
  user-n position tickets...
  .
  ```

  Wenn sich alle nach dem game resource model notwendigen Spieler mit `join` für ein Spiel angemeldet haben, wird dies allen Spielern und Beobachtern mitgeteilt. Es wird außerdem der Status des Spiels wie beim `status`-Kommando angegezeigt. Die Position von Mr.X wird dabei - außer in der Nachricht an Mr.X selbst - durch * ersetzt.

- `syserv: turn <game> <role>`

  Zeigt einem Spieler an, daß er in dem Spiel `<game>` in der Rolle `<role>` am Zug ist.

- `syserv: move <game> <role> <to> <ticket>`

  Zeigt einem Spieler oder Beobachter an, daß im Spiel `<game>` der Spieler mit der Rolle `<role>` nach `<to>` gezogen ist, und dabei das Ticket `<ticket>` benutzt hat. Bei Zügen von Mr.X wird für `<to>` ein `*` übergeben, sofern sich Mr.X in diesem Zug nicht zeigen muß.

- `syserv: leave <game> <role> <reason>`

  Zeigt einem Spieler oder Beobachter an, daß der Spieler mit der Rolle `<role>` das Spiel `<game>` verlassen hat. `<reason>` ist entweder `resign`, wenn der Spieler das Spiel durch ein Kommando verlassen hat, oder `error`, wenn die Verbindung zum dem Spieler durch einen Fehler abgebrochen wurde.

- ```
  syserv: end <game> <reason>
  grmodel
  start-0 start-1 ... start-n
  0 to-0 ticket-0
  1 to-1 ticket-1
  ...
  n to-n ticket-n
  0 to-0 ticket-0
  ...
  n to-n ticket-n
  ...
  0 to-0 ticket-0
  ```

```
...
m to-m ticket-m
.
```

Zeigt den Spielern und Beobachtern eines Spiels dessen Ende an. Ein Spiel ist beendet, wenn die maximale Anzahl von Zügen erreicht ist, Mr.X entdeckt ist, die Detektive nicht mehr ziehen können, oder wenn ein Spieler mit 'resign' das Spiel verläßt. `<reason>` gibt den Grund an, aus dem das Spiel beendet wurde, und ist entweder

`discovered`	wenn Mr.X im letzten Zug entdeckt wurde,
`left`	wenn ein Spieler das Spiel verlassen hat,
`stalled`	wenn alle Detektive nicht mehr ziehen können oder
`escaped`	wenn Mr.X bis zur letzten Runde nicht gefaßt wurde.

Die erste Zeile gibt das game resource model des Spiels an, die zweite die Startpositionen aller Spieler, die weiteren Zeilen beschreiben alle Züge bis zum Ende des Spiels. Mit dieser Information soll ein Client nach dem Ende des Spiels das regelgerechte Ziehen von Mr.X überprüfen.

- `<user>: <msg>`

  Als Folge des `send`-Kommandos können auch Nachrichten von anderen Benutzern des Servers eintreffen.

Fehlercodes

Sowohl Client als auch Server können als Reaktion auf ein empfangenes Kommando, daß nicht erfolgreich bearbeitet werden kann, einen Fehler signalisieren. Das hierfür vorgesehende Kommando lautet

```
<entity>: error <errno>
```

wobei `entity` entweder `syserv` ist, wenn der Server einen Fehler signalisieren möchte bzw. der Username des Spielers, dessen Client einen Fehler signalisiert.

Es existiert eine Reihe von vordefinierten Fehlercodes.

Allgemeine Fehler	
01	Falscher Syntax
02	Unbekanntes Kommando
03	Unbekanntes Game Resource Model
Bei der Spielerschaffung	
10	Benutzer ist bereits registriert
11	Benutzer ist nicht registriert
12	Rolle existiert nicht im Game Resource Model
13	Gewünschte Rolle nicht mehr frei
14	Angegebenes Spiel existiert nicht
15	GRM konnte nicht geparst werden
Ausführen von Spielzügen	
20	Zielfeld nicht erreichbar
21	Angegebens Ticket existiert nicht
22	Spieler verfügt nicht mehr über das Ticket
23	Spieler ist nicht am Zug
24	Spieler ist kein Teilnehmer des Spiels
25	Spieler besitzt die angegebene Rolle nicht
26	Spiel existiert nicht

Beispiel

Hier der Ablauf einer Spielerschaffung aus der Sicht eines Clients:

```
--> syserv: hello

--> register prelle
<-- syserv: ok

--> list grmodels
<-- syserv: grmodels
<-- anfaenger
<-- profi
<-- .

--> get grm anfaenger
<-- \% Der benutzte Stadtplan
<-- MAPNAME london.map
<--
<-- TICKETS
<-- ...
<-- 3 53 10 8 4 0
<-- 4 29 10 8 4 0
<-- .

--> get map london
<-- 3 8 13 18 24
<-- ..
<-- .

--> create testspiel 0 anfaenger
<-- syserv: joined 125
```

Das folgende Beispiel zeigt, wie ein Client sich anmeldet, sich die Liste der verfügbaren Spiele anzeigt und sich dann für eines der Spiele entscheidet, welches daraufhin gestartet wird.

```
<-- syserv: hello

--> register kollmann
<-- syserv: ok

--> list games
<-- syserv: games
<-- testspiel anfaenger london 0
<-- laeuftschon anfaenger bremen 1
<-- .

--> status testspiel
<-- syserv: status testspiel anfaenger london -1
<-- prelle    *  4 3 3 4
<-- Missing 188 10 8 4 0
<-- foo     100 10 8 4 0
<-- bla      53 10 8 4 0
<-- fasel    29 10 8 4 0
```

```
<-- .

--> join testspiel 1
<-- syserv: joined 188

<-- syserv: start testspiel
<-- prelle     *  4 3 3 4
<-- kollmann 188 10 8 4 0
<-- foo      100 10 8 4 0
<-- bla       53 10 8 4 0
<-- fasel     29 10 8 4 0
<-- .
```

Das letzte Beispiel erläutert die Chat-Funktion aus der Sicht des Servers. Dabei sendet `prelle` zuerst eine Nachricht an `kollmann`, dann sendet `kollmann` eine Nachricht an alle Spielteilnehmer und anschließend sprechen sich die vier Detektive untereinander ab.

```
<--prelle    send kollmann Hallo
-->kollmann  prelle: Hallo
-->prelle    syserv: ok

<--kollmann  send testspiel Machen wir 'ne Pause und gehen was Essen?
-->prelle    kollmann: Machen wir 'ne Pause und gehen was Essen?
-->foo       kollmann: Machen wir 'ne Pause und gehen was Essen?
-->bla       kollmann: Machen wir 'ne Pause und gehen was Essen?
-->fasel     kollmann: Machen wir 'ne Pause und gehen was Essen?
-->kollmann  syserv: ok

<--foo       send kollmann,bla,fasel Ich glaube Mr.X steht auf 200
-->kollmann  foo: Ich glaube Mr.X steht auf 200
-->bla       foo: Ich glaube Mr.X steht auf 200
-->fasel     foo: Ich glaube Mr.X steht auf 200
-->foo       syserv: ok
```

Hinweise

- Gestaltet Eure Protokoll-Engine robust. Sie sollte nicht aus dem Tritt kommen, wenn zwischen zwei Argumenten mehr als ein Leerzeichen steht, irgendwo Groß- statt Kleinbuchstaben vorkommen oder wenn eine Dezimalzahl mit führen Nullen angegeben wird.
- Server und Client sollten auf unbekannte Kommandos mit einer entsprechenden Fehlermeldung reagieren, daß Spiel aber nicht abbrechen.
- Es mag Definitionslücken in den Kommandos geben. Wenn Ihr glaubt welche gefunden zu haben, benutzt die Newsgroup `fb3.lv.swp`, um dies zu diskutieren.
- Die Newsgroup ist auch ein guter Platz, wenn Ihr weitere Features oder Fehlercodes einführen wollt und eure Ideen/Erweiterungen veröffentlichen wollt.
- Bedenkt bitte, daß Eure Programme mit dem hier beschriebenen minimalen Kommandosatz arbeiten können sollten, selbst wenn ihr weitere Erweiterungen implementiert habt.
- Am Abschluß des Softwarepraktikums würden wir gerne Clients und Server verschiedener Gruppen mixen, um damit eure Implementierungen zu testen und etwas zu spielen. Achtet daher auf Kompatibilität.