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

15. Mai 2001

Softwarepraktikum 2001
Scotland Yard Protokollbeschreibung V.1.1

1 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.

1.1 Änderungen in dieser Version

Aufgrund der Diskussionen in der Newsgroup sind einige Unklarheiten und Schwächen des Protokolls zutage gekommen. Dieses Dokument enthält nun eine überarbeitete Fassung dieses Protokolls - teilweise auf Kosten der Kompatibilität.
Wichtig: Für euren Entwurf und eure Implementierung solltet ihr explizit angeben, wenn welche Protokollversion ihr verwendet.

Hier zunächst eine Kurzübersicht über erfolgente Änderungen:

- Im `send`-*Kommando* ist jetzt definiert, daß die Empfängerliste keine Leerzeichen enthalten darf (wie bereits in Newsgroup festgelegt) und daß ein Doppelpunkt am Anfang eines Empfängernamens ein Spielnamen kennzeichnet (ohne selber Bestandteil des Namens zu sein)
- Es gab Änderungen an den Fehlercodes:
  - Die Zuteilung von Fehlern zu Situation entfällt
  - *Fehlercode 26* entfällt, da keine Abgrenzung mehr zu Fehler 14 vorhanden
  - Es sind neue Fehlercodes dazugekommen
  - Eine Fehlermeldung kann jetzt weitere Parameter haben
- Das `get`-*Kommando* erklärt jetzt, daß anhand der Endung des Namens unterschieden wird, ob es sich um ein GRM oder eine Karte handelt.
  Es ist ferner definiert, daß beim Übertragen eines GRMs die Position von Mr.X durch ein * ersetzt werden sollte (aber nicht muß)
- Neue Kommando `ping` und `pong` (nicht lachen) hinzugefügt, um die Lebendigkeit einer Verbindung prüfen zu können.

- Aufgeführt, welches synchrone Kommando welche Fehler produzieren kann. (Mag sein, daß da noch welche nicht bedacht wurden → Newsgroup)
- `hello`-Kommandos um eine Protokollinformation ergänzt.
- Beispiele erweitert und korrigiert
- Erläuterungen zum Problem der Kommando-Überlappung
- Kleinere Korrekturen und Ergänzungen in den Kommando-Texten - insbesondere `move` und `send`.

Wer bisher die Diskussion in der Newsgroup verfolgt hat, sollte keine größeren Neuigkeiten hier finden. Kompatibilitätsprobleme sollte es daher keine geben, mit Ausnahme des geänderten Satzes der Fehlercodes - aber selbst daß sollte keine größeren Auswirkungen haben, insbesondere auf den normalen Spielablauf.

1.2 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>
.
```

1.2.1 User- und Spielbezeichner

Der Name eines Users oder eines Spieles darf nur aus Buchstaben, Ziffern und dem Unterstrich bestehen ($[a..zA..Z\_]$).

1.2.2 Kommandoüberlappung

Eine mögliche Schwierigkeit die man haben kann, ist die, daß man auf die Antwort eines synchronen Kommandos wartet, und plötzlich ein asynchrones Kommando erhält und die eigentliche Antwort erst kurz darauf.
Ein Beispiel: Client A sendet eine Anfrage an den Server (z.B. get london.map) und wartet nun auf die Reaktion des Servers (syserv map london ...). Zeitgleich hat Client B versucht, Client A eine Chat-Nachricht zu senden (send <userA> Hallo), was dazu führt, daß der Server ein userA: Hallo sendet.
Das Problem ist nun, daß theoretisch bei Client A die Chat-Nachricht VOR der Reaktion auf das Kommando eintreffen kann.

Bedenkt bei euren Protokollimplementierungen, daß asynchrone Kommandos fast jederzeit dazwischenfunken können. Am besten baut ihr eine Art State-Machine, die ihren Zustand durch asynchrone Kommandos nicht verändert.

1.3 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> <parameter ....>
```

wobei <errno> ein Dezimalzahl ist, die die Fehlerursache beschreibt.
ACHTUNG! Auch wenn in der folgenden Aufzählung nicht extra aufgeführt, können die Fehlermeldungen 1, 2 und 6 jederzeit auftreten.

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

1. **register**

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

Syntax	register <name>
Reaktion	syserv: ok
Fehler	10

2. **help**

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

Syntax	help [<command>]
Reaktion	syserv: help Hilfe-Text ... Hilfe-Text .
Fehler	-

3. **exit**

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

Syntax	`exit`
Reaktion	`syserv: ok`
Fehler	-

4. **who**

   Liste aller registrierten Benutzer - eine Zeile für jeden Benutzer - anzeigen. Jede Zeile enhält den Namen des Benutzers und evtl. die IP-Adresse bzw. den Hostnamen, 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.

Syntax	who	who host	who addr
Reaktion	syserv: users user-1 user-2 ... user-n .	syserv: users user-1 hostname-1 user-2 hostname-2 ... user-n hostname-n .	syserv: users user-1 ipaddress-1 user-2 ipaddress-2 ... user-n ipaddress-n .
Fehler	-		

5. **send**

   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.

Syntax	send <user-list> <message>
Reaktion	syserv: ok
Fehler	11,14

   Um beim Parsen die User-List von der eigentlichen Nachricht unterscheiden zu können, sind Leerzeichen innerhalb der Userlist nicht erlaubt.
   Ob es sich bei einem Namen um einen User oder ein Spiel handelt, wird dadurch erkannt, daß Spielnamen ein Doppelpunkt vorangestellt wird.
   Ist einer der angegebenen Empfänger ungültig, liegt es im Ermessen des Servers, ob er in diesem Fall einen Fehler melden möchte. Ein Beispiel:

```
client--->server: send prelle,wolf_dietrich,:einSpiel Hallo ihr da draußen
client<---server: syserv: error 11 wolf_dietrich
client--->server: send prelle,:einSpiel Hallo ihr da draußen
client<---server: syserv: ok
```

   Es ist übrigens nicht definiert, ob es möglich ist, sich selber als Empfänger anzugeben (direkt oder indirekt über Spielnamen). Das liegt in eurem Ermessen.

6. **create**

   Neue Spiele werden mit diesem Kommando unter einem neuen Namen und mit dem angegebenen game resource model eröffnet.

Syntax	create <game> <role> <grmodel>
Reaktion	syserv: joined <position>
Fehler	03,12,15,16

   <role> gibt an, in welche Rolle der Spieler übernehmen möchte. Dabei bedeutet:

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

   usw.

Es 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.

7. **join**

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

Syntax	`join <game> <role>`
Reaktion	`syserv: joined <position>`
Fehler	12,14

8. **move**

Dieses Kommando wird sowohl vom Client, als auch vom Server verwendet, um einen ausgeführten Zug anzuzeigen.
Wenn es von einem Client an den Server gesendet wird, dient es dazu, mitzuteilen, welchen Zug man ausführen möchte und erfolgt immer als Reaktion auf ein `turn` (S.8). Der Zug wird vom Server geprüft und evtl. mit einem Fehler abgewiesen.
Wird es vom Server zu einem Client gesendet (als asynchrones Kommando), dient es dazu, mitzuteilen, welchen Zug ein anderer Spieler in diesem Spiel gemacht hat. In diesem Fall darf ein Client keine Fehler melden. Hier gilt auch die Sonderregelung, daß für `to` ein `*` eingesetzt werden muß, wenn Mr.X gezogen ist und sich nicht zeigen mußte.
Zur Klärung: Wenn ein Spieler einen Zug durchführt, informiert der Server nur die anderen Spieler und die Zuschauer über den erfolgten Zug (siehe Beispiel 2.4).

Syntax	`move <game> <role> <to> <ticket>`
Reaktion	`syserv: syserv ok`
Fehler	14,20,21,22,23,24,25 (nur vom Server)

9. **resign**

Ein laufendes Spiel kann verlassen werden. Nachdem ein Spieler das Spiel verlassen hat, informiert der Server die übrigen Clients mit einem `syserv: end <game> left` über das vorzeitige Spielende.

Syntax	`resign <game>`
Reaktion	`syserv: ok`
Fehler	14,24

10. **watch**

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

Syntax	`watch <game> [on|off]`
Reaktion	`syserv: ok`
Fehler	14

11. **status**

Zeigt den Status eines Spiels an.

Syntax	`status <game>`

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

Fehler	14

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**

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.

Syntax	`list grmodels [<map>]`

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

Fehler	-

13. **list maps**

Zeigt die Liste der vorhanden Stadtpläne an.

Syntax	`list maps`

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

Fehler	-

14. **list games**

Zeigt die Namen der eröffneten Spiele an.

Syntax	`list games`

Reaktion	`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`
	`.`

Fehler	-

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**

Gibt dem Client die Möglichkeit, sich Maps und GRMs vom Server zu besorgen. Ob es sich um eine Karte oder um ein GRM handelt wird anhand der Dateiendung erkannt (`.map` oder tt .grm.
Um schummelnde Clients unmöglich zu machen, sollte im übertragenen GRM die Startposition von Mr.X immer durch ein * ersetzt werden - das ist aber optional.

<table>
<tr><td>Syntax</td><td><code>get <mapname>.map</code></td><td><code>get grmodel <grmname>.grm</code></td></tr>
<tr><td>Reaktion</td><td><code>syserv: map <map></code>
<code>map-text</code>
<code>...</code>
<code>map-text</code>
<code>.</code></td><td><code>syserv: grmodel <grmodel></code>
<code>grmodel-text</code>
<code>...</code>
<code>grmodel-text</code>
<code>.</code></td></tr>
<tr><td>Fehler</td><td>04</td><td>03</td></tr>
</table>

1.4 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:

1. **hello**

   Diese Nachricht wird einmal sofort nach dem Aufbau der Verbindung vom Server zum Client geschickt. Clients, die die überarbeitete Protokollversion unterstützen, sollten diese Nachricht ebenfalls mit einem `hello` beantworten.

Syntax	`syserv: hello [<version>]`
Reaktion	hello <version>
Fehler	05

2. **ping**

   Um zu Prüfen, ob eine TCP-Verbindung noch aktiv ist, ist dieses Kommando gedacht. Sowohl Client, als auch Server sollten diese Nachricht jederzeit senden können. Der Empfänger muß die Nachricht mit einem `pong` beantworten.
   Anmerkung: Ein `ping` an eine Komponente mit dem alten Protokoll sollte immer einen Fehler 02 produzieren - also auch eine Art Antwort, auf die man warten kann.

Syntax	`ping`
Reaktion	`pong`
Fehler	-

3. **start**

   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.

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

Reaktion	-
Fehler	-

4. **turn**

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

Syntax	`syserv: turn <game> <role>`
Reaktion	move ...
Fehler	-

5. **move**

   Vergleichbar mit dem synchronen `move`-Kommando (S.5), jedoch wird bei Zügen von Mr.X für `<to>` ein `*` übergeben, sofern sich Mr.X in diesem Zug nicht zeigen muß.

6. **leave**

   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.

Syntax	`syserv: leave <game> <role> <reason>`
Reaktion	-
Fehler	-

7. **end**

<table>
<tr><td>Syntax</td><td><pre>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
.</pre></td></tr>
<tr><td>Reaktion</td><td>-</td></tr>
<tr><td>Fehler</td><td>-</td></tr>
</table>

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.

8. **<user>:**

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

Syntax	`<user>: <message>`
Reaktion	-
Fehler	-

1.5 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> [<parameter>]
```

wobei `entity` entweder `syserv` ist, wenn der Server einen Fehler signalisieren möchte bzw. der Username des Spielers, dessen Client einen Fehler signalisiert. Bei Bedarf kann ein weiterer Parameter hinzugefügt werden, der näheren Hinweis auf die Ursache des Fehlers gibt. Dieser weitere Parameter ist der Einfachheit halber alles, was der Fehlernummer folgt.

Es existiert eine Reihe von vordefinierten Fehlercodes.

Nr.	Bedeutung	Parameter
01	Falscher Syntax	-
02	Unbekanntes Kommando	Kommandoname
03	Unbekanntes Game Resource Model	GRM Name
04	Unbekannter Kartenname	Mapname
05	Unsupported Protocolversion	-
06	Unbekannter Fehler	Fehlertext
10	Benutzer ist bereits registriert	Username
11	Benutzer ist nicht registriert	Username
12	Rolle existiert nicht im Game Resource Model	Nr. der Rolle
13	Gewünschte Rolle nicht mehr frei	Nr. der Rolle
14	Angegebenes Spiel existiert nicht	Spielname
15	GRM konnte nicht geparst werden	-
16	Spiel existiert bereits	-
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	-

2 Beispiele

2.1 Ein Spiel erschaffen

Hier der Ablauf einer Spielerschaffung aus der Sicht eines Clients:

```
<-- syserv: hello 1.1
--> hello 1.1

--> register prelle
<-- syserv: ok

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

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

--> get map london
<-- 1 46 underground
<-- 1 58 bus
<-- 1 8 taxi
<-- ...
<-- .

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

2.2 Bei einem Spiel anmelden

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

```
<-- syserv: hello 1.1
--> hello 1.1

--> register prelle
<-- syserv: 10 prelle
```

```
--> 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
<-- .
```

2.3 Spiel starten

Irgendwann sind alle Rollen besetzt, der Server startet das Spiel und teilt Mr.X (prelle) mit, daß er am Zuge ist.

Zunächst bekommen alle Spieler folgende Nachricht:

```
<-- 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
<-- .
```

2.4 Spielen

Dann geht es z.B. wie folgt weiter:

```
-->prelle    syserv: turn testspiel 0
<--prelle    move testspiel 0 126 taxi
-->prelle    syserv: ok

-->kollmann  syserv: move testspiel 0 * taxi
-->...       syserv: move testspiel 0 * taxi

-->kollmann  syserv: turn testspiel 1
<--kollmann  move testspiel 1 190 bus
-->kollmann  syserv: ok
```

```
-->prelle    syserv: move testspiel 1 190 bus
-->...       syserv: move testspiel 1 190 bus
```

2.5 Prüfen der Verbindung

Zwischenzeitlich war längere Zeit Ruhe auf der Leitung, so daß z.B. der Server sich genötigt sieht, die Verbindung zu den Clients zu prüfen:

```
-->prelle    syserv: ping
<--prelle    pong
```

Käme ein Client auf die gleiche Idee, sähe das so aus:

```
<--prelle    ping
-->prelle    syserv: pong
```

2.6 Chat

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
```

3 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.