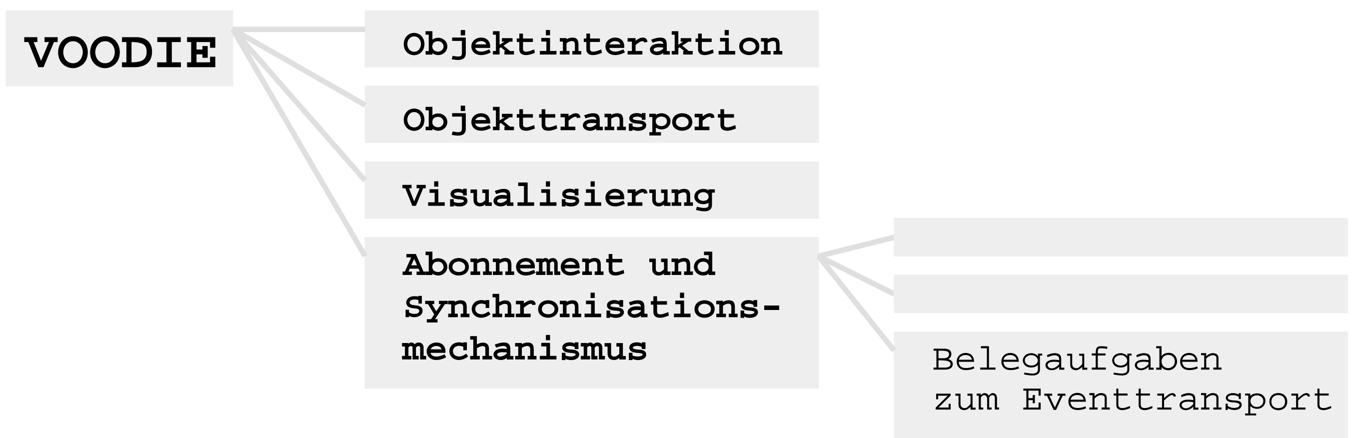
Virtual **O**bject **O**riented **D**istributed **I**nteractive **E**nvironment

ein Kommunikationsmechanismus für verteilte virtuelle Umgebungen

Gliederung

Einführung
der Mechanismus
die derzeitige Implentation
die Aufgaben

Einführung



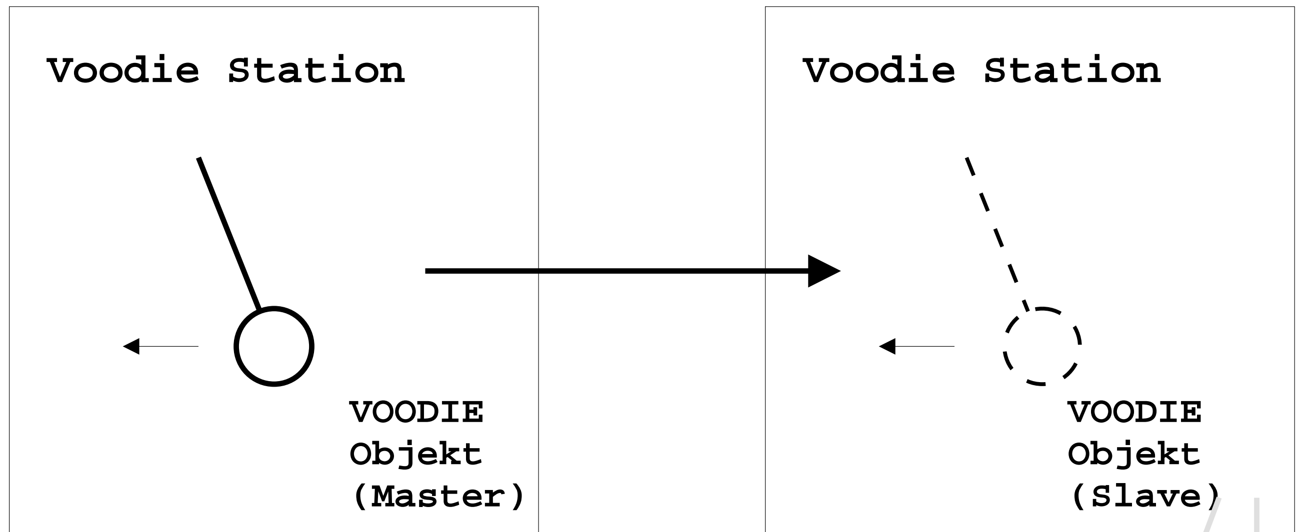
Teiaspekte des VOODIE systems

Das VOODIE (Virtual Object Oriented Interactive Environment) System stellt eine verteilte virtuelle Umgebung dar, in der verschiedene Objekte dezentral vorhanden sind.
Um den Teilnehmern der virtuellen Umgebung eine gemeinsame Sicht auf diese Objekte zu ermöglichen, müssen die Objekte u.U. dupliziert werden. Ein verteiltes VOODIEobjekt kann also aus mehreren duplizierten Objekten bestehen. Dabei besitzt jeweils ein Objekt der so entstehenden Objektgruppe die Objektkontrolle (Masterobjekt).
Der Mechanismus stellt Funktionalitäten zur Verfügung, die es ermöglichen, solche Objektgruppen zu erstellen, die einzelnen Objekte zu synchronisieren und die Objektkontrolle von einem zu einem anderen Objekt zu übertragen.
Die Kommunikation der Objekte innerhalb der Objektgruppen erfolgt dabei über Ereignisse (Events).

Die Aufgaben die im Rahmen dieser Lehrveranstaltung zu lösen sind, beziehen sich auf die Übertragung dieser Events über das Netzwerk.

Belegaufgaben zur Lehrveranstaltung Rechnernetze II
Marko Meister, april 1997

der Mechanismus (I)

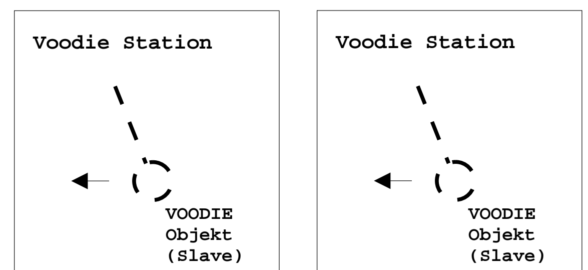


VOODIE besteht aus einzelnen Stationen, auf denen Objekte existieren.

Objekte einzelner Stationen können von anderene Stationen aboniert werden

Es entstehen Objektgruppen, die aus einem Masterobjekt und beliebig vielen Slaveobjekten bestehen.

Synchronisation der Objektgruppe durch UPDATE Events von Master an Slave bzw. Abomaster an Slave (über Netzwerk)

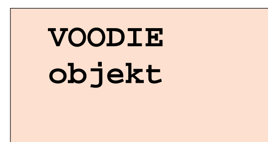


der Mechanismus (II)

VOODIE Station 1
Master Objekt

VOODIE Station 2
Slave Objekt

Phase 1:
OI-Übertragung

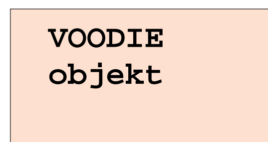


SEND_ALL_OI Event

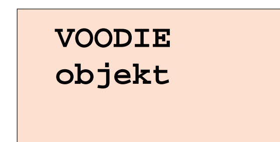
OI Event

Generiere
Slave Objekt
aus OI-Infos

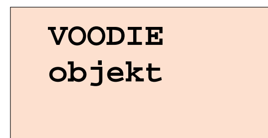
Phase 2:
Abonierung



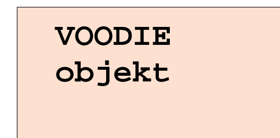
SUBSCRIBE Event



Phase 3:
Synchroni-
sation



UPDATE Events



die derzeitige Implementierung (I)

Die derzeitige Applikation wurde unter Zuhilfenahme von POSIX-Threads implementiert. Dabei laufen die Module in seperaten Threads ab, d.h. die Ausführung der einzelnen Module erfolgt quasiparallel.
Alle Threads einer Applikation laufen in ein und demselben Prozess ab. Deshalb ist die Kommunikation zwischen den einzelnen Modulen sehr viel einfacher möglich, als bei der Benutzung von verschiedenen Prozessen.

Das POSIX Thread Interface (pthreads) stellt eine einfache API für die Benutzung von Threads in Applikationen zur Verfügung.

Prinzipiell werden alle Module als separate Funktionen implementiert. Diese Funktionen werden dann über einen pthread_create() Aufruf als separate Threads gestartet.

Die Kommunikation der Module untereinander kann dabei über globale Variablen erfolgen. Die derzeitige VOODIE Implementation benutzt globale Eventqueues zur Kommunikation.

Der Zugriff Aufschlag die gemeinsam benutzten Daten muß natürlich synchronisiert werden. Die POSIX Thread API stellt dafür eine Reihe von Mechanismen zur Verfügung.

Beispielhaft soll an dieser Stelle die Benutzung einer MUTEX Variable vorgestellt werden.

die derzeitige Implementierung (II)

```
/* Test program fuer pthreads
 Makefile looks like this:
#
# Makefile for pthreads test Program
# Author: Marko Meister
#
CC  = cc
CFLAGS=-fullwarn -xansi -D_REENTRANT -DPTHREADS
LFLAGS=-lpthread
default: test
clean:
     rm -f *.o test1 core
test:  test.c Makefile
     $(CC) $(CFLAGS) -o test.o -c test.c
     $(CC) $(CFLAGS) -o test test.o $(LFLAGS)
*/

#include <pthread.h>
#include <stdio.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int i = 0;

void mache_etwas (void) {
  pthread_mutex_lock(&mutex);
  i++;
  printf("mache_etwas: habe Mutex, i=%d!\n",i);
  pthread_mutex_unlock(&mutex);
  return;
}
void mache_etwas_anderes(void) {
  pthread_mutex_lock(&mutex);
  i++;
  printf("mache_etwas_anderes: habe Mutex, i=%d!\n",i);
  pthread_mutex_unlock(&mutex);
  return;
}

int main(int argc, char** argv) {
  pthread_t thread1, thread2;

  pthread_create (&thread1, NULL, (void *)  mache_etwas, NULL);
  pthread_create (&thread2, NULL, (void *)  mache_etwas_anderes, NULL);

  pthread_mutex_lock(&mutex);
  i++;
  printf("main: habe die Mutex, i=%d!\n",i);
  pthread_mutex_unlock(&mutex);

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);

  printf("main: fertig, i=%d!\n",i);

  return (0);
}
```

die derzeitige Implementierung (III)

Ausgabe:

```
mache_etwas: habe die Mutex, i=1!
mache_etwas_anderes: habe die Mutex, i=2!
main: habe die Mutex, i=3!
main: fertig i=3!
```

oder:

```
mache_etwas_anderes: habe die Mutex, i=1!
mache_etwas: habe die Mutex, i=2!
main: habe die Mutex, i=3!
main: fertig i=3!
```

oder:

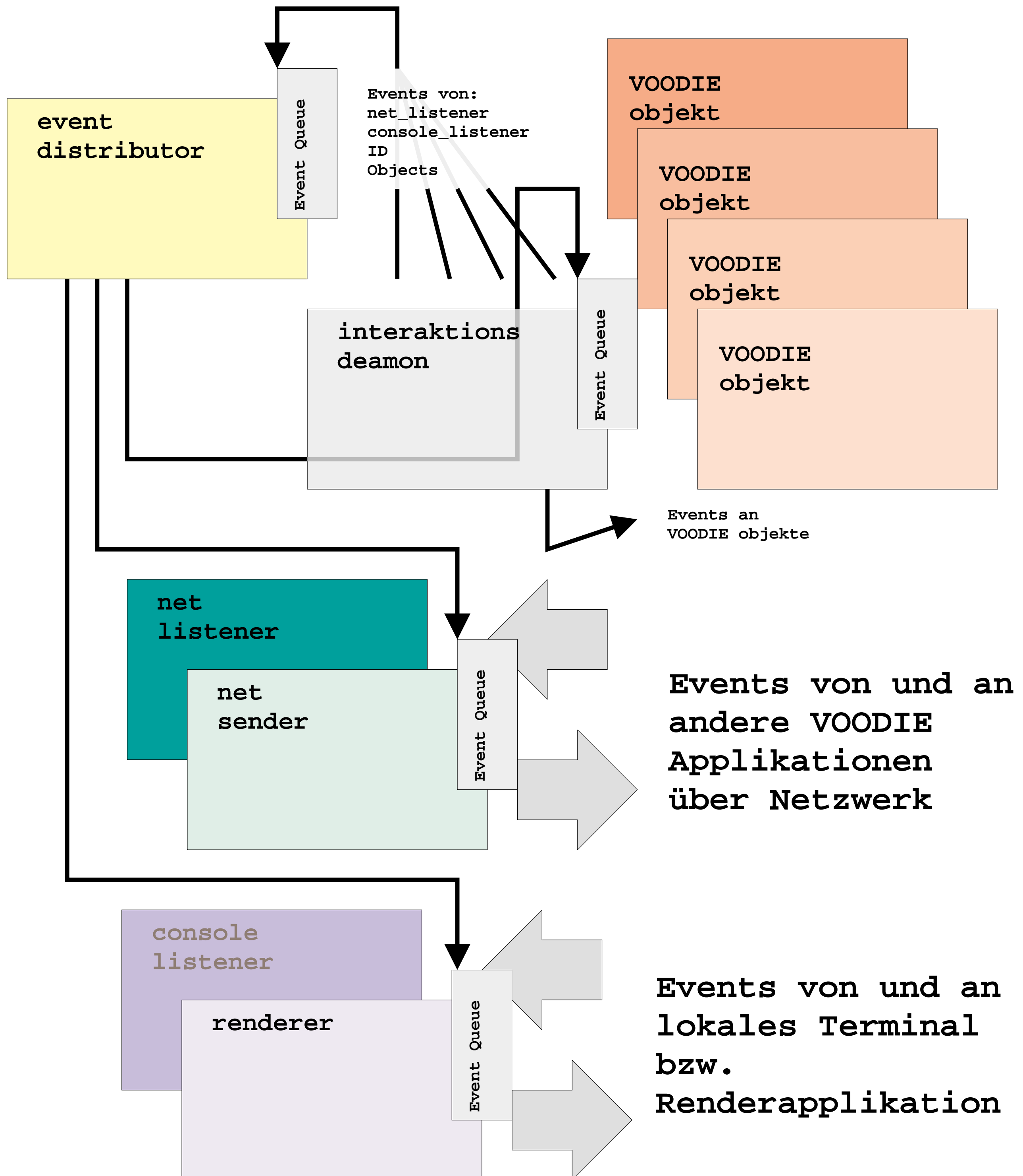
```
main:: habe die Mutex, i=1!
mache_etwas_anderes: habe die Mutex, i=2!
mache_etwas: habe die Mutex, i=3!
main: fertig i=3!
```

oder anders ...

Literatur:

B. Nichols, D. Buttlar, J. Proulx Farrwell:
Pthreads Programming; O'Reilly & Associates 1996;
BIB: 24.542.- 217

die derzeitige Implementierung (IV)



die Aufgaben

(1) Low level protocol (Eventtransfer) redesign 1
Das einfache Protokoll zur Uebertragung von events soll durch ein effizienteres Verfahren ersetzt werden.
Insbesondere sind folgende Hinweise zu beachten:
- Effizienz und Sicherheit
- minimaler Aufwand acknowledgement
- intelligentes Verbindungsmanagement
- Aufwandsanalyse fuer uebertragungsphase (connect, send, recv, close ...)

Die Ergebnisse sind anhand von Messdaten zu belegen.

(2) Performance-Tests fuer low-level Protocol
Fuer das Projekt ist eine geeignete Testumgebung zur Performanceanalyse zu entwickeln.
Im Rahmen der Spezifikation sind insbesondere geeignete Maszstaebe fuer die Performancemessung festzulegen. Die Ergebnisse sollen anhand graphischer Darstellungen ausgewertet werden.

(3) non_blocking fuer Kommunikationsmodule
Die Empfangsmodule sind gegenwaertig lediglich zum Empfang eines events in der Lage. Die Module sollen so modifiziert werden, dass mehrere Verbindungen gleichzeitig entgegenzunehmen sind. Die Realisierung sollte mit pthreads erfolgen, die Variante mit fork()-Rufen sollen zur Effizienzuntersuchung gegenuebergestellt werden.

(4) error handling
Die Empfangs- und Sendemodule sind um Mechanismen zur Fehlerbehandlung zu erweitern. Die Fehlerursachen sind systematisch zusammenzustellen.
Es sind geeignete Mechanismen in die Module zu integrieren, die verschiedene Fehler (z.B. fehlendes EOM), Verbindungsabbruch waehrend der Uebertragung oder timeouts robust erkennen und bearbeiten koennen.

(5) Low level (Eventtransfer) protocol redesign 2
Durch streambasierte Datenuebertragung wird gegenwaertig die Uebertragung unbegrenzter Eventgroessen unterstuetzt.
Es ist ein datagramm-basierter Eventtransport zu implementieren.
Dabei soll die zulaessige Eventgroesse festgestellt werden. Zusaetzlich soll ein Mechanismus vorgesehen werden, der die zuverlaessige Uebertragung von Events sicherstellt.
Fuer die datagramm-Uebertragung sind geeignete statistische Performanceanalysen durchzufuehren.

(6) multicasting
Es sind Grundsatzueberlegungen zur Realisierung eines Multicastbetriebes zur Objektsysnchronisation anzustellen und prototypmaessig zu implementieren.

(7) TLI
Die Protokollimplementierung der Basisfunktionen (send_message(), recv_message() (vgl. Funktionsinterface) soll SVR4-konform auf Basis der TLI-Rufe erfolgen.
Performanceunterschiede sind zu analysieren. Aussagen zur

Bemerkungen

```
Low level (Eventtransfer) Protocol
(Eventlänge unbegrenzt (Streamvariante)):

   LISTENER : erwarte Verbindung
   SENDER   : stelle Verbindung her
   LISTENER : sende Begrüssung
                This is VOODIE <Version>
                accepting Events:
                (TYPE FROM_HOST FROM_MODULE
                 TO_HOST TO_MODULE MESSAGE end of message) ...
   SENDER   : lese Begrüssung
   wiederhole
          SENDER   : sende ein Paket (ein Teil (einige Bytes)
                     der Zeichenkette)
          LISTENER : lese ein Paket
          LISTENER : sende Empfangsbestätigung
                     received <num> Bytes. ok!
          SENDER   : erwarte und lese Empfangsbestätigung
          LISTENER : hänge das Paket an die bereits
                     empfangene Zeichenkette an
   bis die gesamte Zeichenkette übertragen wurde.
   SENDER   : sende end of message
   LISTENER : sende Bestätigung, Verbindung abbrechen
   SENDER   : lese Bestätigung, Verbindung abbrechen
   LISTENER : empfangene Zeichenkette ist Rückgabewert
```

Bemerkungen

(2) Die Events in Voodie koennen als nullterminierte Zeichenketten angesehen werden.

(3) als Portnummer soll DVR_PORT_NUM benutzt werden. Dieser Wert sollte so definiert sein:

```
#define  DVR_NET_PORT_NUMBER 3490
```

(4) Es sollen unbegrenzt grosse Zeichenketten moeglich sein (ausser bei Aufgabe 5). Die maximale Laenge der Zeichenketten soll zur Compilezeit so festgelegt werden:

```
#define  DVR_NET_MAX_MSG_SIZE 100000
     // maximal 100Kbyte lange Zeichenketten
```

(5) Funktionsinterfaces:

(a) Zeichenkette (Event) senden (Sender):

```
int send_message(const char* to_host,
                 const char* message)

            returnvalue = 0 on succsess
            returnvalue != 0 on error
```

Bemerkungen:
- gegebenenfalls sind Rueckgabewerte zu definieren, die die Art des aufgetretenen Fehlers spezifizieren.
- to_host enthaelt eine IP Adresse als String (dotted quad bzw. fqdn)

(b) Zeichenkette (Event) empfangen

```
int recv_message(char* from_host, char* message)

            returnvalue = 0 on succsess
            returnvalue != 0 on error
```

Bemerkungen:
- gegebenenfalls sind Rueckgabewerte zu definieren, die die Art des aufgetretenen Fehlers spezifizieren.
- from_host enthaelt (nach dem Aufruf) eine IP Adresse als String
- der Platz fuer from_host und message ist durch die Routine zu alloziieren (new char[needed_lenght], wobei needed_lenght der benoetigten Anzahl Zeichen entspricht).

(6) Existierende Funktionen
Zur Bearbeitung der Aufgaben koennen bei Bedarf funktionsfaehige send_message() und recv_message() Funktionen zur Verfuegung gestellt werden.