Aufgabe Hausarbeit

In dieser Hausarbeit wird die Open Source Sprache Erlang/OTP verwendet (Plugin für Eclipse, auch für andere Editoren gibt es eine Unterstützung, etwa Notepad++, KWrite, XEmacs etc.). Das System demo ist speziell dem Austausch von Nachrichten und einem RPC gewidmet.

Lesen Sie sich die Aufgabenstellung sorgfältig durch!

"Übergaben" an Datenstrukturen von Erlang OTP sind untersagt. Etwa die Verwendung von z.B. `dict` oder `sets`. Lediglich die Basis-Strukturen Liste (`lists`) und Tupel (`tuple`) dürfen eingesetzt werden (Funktionen der zugehörigen Module dürfen nicht eingesetzt werden). Algorithmen auf diesen Basisstrukturen müssen selbst implementiert werden! Bei z.B. dem Modul `lists` kann lediglich `[ . | . ]` genutzt werden. Zur Unterstützung steht die Datei util.erl zur Verfügung. Fragen Sie ggf. per E-Mail nach, ob Sie Erlang-Funktionen in der vorgesehenen Art einsetzen dürfen. Bei Einsatz nicht erlaubter Funktionen wird die Abgabe als fehlerhaft gewertet.

Das Client-Server-System wird Ihnen zur Verfügun gestellt: CSsystem. Achtung: das CSsystem entbindet Sie nicht von eigenen Tests und sichert keine korrekte Abgabe zu! Die in Kombination mit Ihrer Implementierung erstellten log-Dateien (einmal Ihre DLQ und das CSsystem, einmal Ihre HBQ und das CSsystem sowie Ihre HBQ und DLQ und das CSsystem) **sind der Abgabe bitte mit beizufügen**. Zudem wird Ihnen das Modul hbqF als Erlang-Datei zur Verfügung gestellt. Damit können Sie die HBQ manuell über Funktionen ansprechen.

Delivery and Holdback Queue

Implementieren Sie in Erlang/OTP ADTs für eine Client/Server-Anwendung; ein Server verwaltet Nachrichten (Textzeilen und Verwaltungsinformationen): **die Nachrichten des Tages**, die von unterschiedlichen Redakteuren (in einem Client-Programm enthalten) ihm zugesendet werden. Diese Nachrichten sind eindeutig nummeriert. Um eine korrekte Nummerierung bei der Auslieferung zu gewährleisten, verwendet der Server das ADT-Konzept einer Delivery und Holdback Queue. Die Leser (in einem Client-Programm enthalten) fragen in bestimmten Abständen die aktuellen Nachrichten ab. Damit ein Leser nicht immer alle Nachrichten erhält, erinnert sich der Server an ihn und welche Nachricht er ihm zuletzt zugestellt hat. Meldet sich dieser Leser jedoch eine gewisse Zeit lang nicht, so vergisst der Server diesen Leser.

Funktionalität

HBQ/DLQ

1. Die **Nachrichten** werden vom Server durchnummeriert (beginnend bei 1) und stellen eine eindeutige ID für jede Nachricht dar.
2. Da die dem Server zugestellten Nachrichten bzgl. der Nummerierung in zusammenhängender Reihenfolge erscheinen sollen und Nachrichten verloren gehen können bzw. in nicht sortierter Reihenfolge eintreffen können, arbeitet der Server intern mit einer **Delivery Queue** (DLQ) und einer **Holdback Queue** (HBQ).
   In der **Delivery Queue** stehen die Nachrichten, die an die Leser ausgeliefert werden können, maximal *?Xdlq* viele Nachrichten. *?Xdlq* wird als die Größe der Delivery Queue bezeichnet.
   In der **Holdback Queue** stehen alle empfangenen Nachrichten, die nicht ausgeliefert werden dürfen. Die Nachrichten der HBQ werden regelmässig auf Auslieferbar geprüft und dann ggf. in die DLQ verschoben. Nachrichten der Redakteure werden daher prinzipiell zunächst in der HBQ gespeichert.
3. Wenn in der HBQ von der Anzahl her mehr als 2/3-tel an Nachrichten enthalten sind, als durch die vorgegebene maximale Anzahl an Nachrichten in der DLQ (*?Xdlq*) stehen können, dann wird, sofern eine Lücke besteht, diese Lücke zwischen DLQ und HBQ mit **genau einer Fehlernachricht** geschlossen, etwa: *"***Fehlernachricht fuer Nachrichtennummern 11 bis 17 um 16.05 18:01:30,580*", indem diese Fehlernachricht in die DLQ eingetragen wird und als Nachrichten-ID die größte fehlende ID der Lücke erhält (im Beispiel also 17). Es werden zunächst keine weiteren Lücken innerhalb der HBQ behandelt, da das System nach Generierung der Fehlernachricht zunächst in den normalen Zustand zurück kehrt! In dem Sinne wird die HBQ in diesem Fall nicht zwingend geleert. Die Fehlernachricht ist nur durch eine entsprechende Zeichenkette in der Nachricht zu erkennen. Ansonsten hat sie das Format einer ganz normalen Nachricht, d.h. das System kann eine Fehlernachricht nach Speicherung in der DLQ nicht mehr als solche erkennen!
4. Der Server verwendet unter anderem zwei ADTs: HBQ (Datei hbq.erl) und DLQ (Datei dlq.erl). Diese dürfen hauptsächlich nur als Erlang-Liste (`[ ]`) realisiert werden! Als Hilfsstrukturen dürfen Tupel (`{ }`) eingesetzt werden. Dazu sind die weiter unten aufgeführten Vorgaben zu beachten! ACHTUNG: diese Dateien beschreiben jeweils alleine die ADT und müssen über alle Implementierungen austauschbar sein (weitere Dateien sind nicht zulässig!)
5. Die HBQ ist als entfernte ADT zu implementieren. Ihre Schnittstelle ist daher durch Nachrichtenformate beschrieben. Die DLQ ist als lokale ADT zu implementieren. Daher sind ihre Schnittstellen durch Funktionen beschrieben. Intern

kann die DLQ jedoch auch als entfernte ADT realisiert werden!

GUI

- **HBQ-DLQ-GUI**: Die Ausgaben sind alle in eine Datei HB-DLQ<Node>.log zu schreiben: ein Beispiel ist im CSsystem zu finden.

Fehlerbehandlung

15. Das loggen der Ausgaben sollte mögliche Fehler leicht auffindbar machen.

Schnittstellen

Im Folgenden wird die **Schnittstelle der HBQ** des Servers beschrieben.

```
/* Starten der HBQ auf der Node der HBQ (oder entfernt starten) */
initHBQ(DLQ-Limit,HBQ-Name)

/* Speichern einer Nachricht in der HBQ */
HBQ ! {self(), {request,pushHBQ,[NNr,Msg,TSclientout]}}
receive {reply, ok}
Beispiel:  HBQ ! {<7016.50.0>,  {request,pushHBQ,[15,  "0-pclient@KI-VS-<0.64.0>-KLC: 15te_Nachricht. C
Out: 29.04 08:43:24,871| ", {1430,289804,872001}]}}
        receive {reply, ok}

/* Abfrage einer Nachricht */
HBQ ! {self(), {request,deliverMSG,NNr,ToClient}}
receive {reply, SendNNr}
Beispiel:  HBQ ! {<7016.50.0>, {request,deliverMSG,15,<7298.65.0>}}
        receive {reply, 18}

/* Aktueller Stand der ADT in log schreiben */
HBQ ! {self(),{request,listDLQ}}
receive {reply, ok}
Beispiel:  HBQ ! {<7016.50.0>, listDLQ}
            dlq>>> Content(37):
[37,36,35,34,33,32,31,30,29,28,27,26,25,24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]
HBQ ! {self(),{request,listHBQ}}
receive {reply, ok}
Beispiel:  HBQ ! {<7016.50.0>, listHBQ}
            HBQ>>> Content(28):
[474,473,472,466,465,464,463,462,461,460,459,458,457,456,455,454,453,452,451,450,449,448,444,443,442,441,438,434]

/* Terminierung der HBQ */
HBQ ! {self(), {request,dellHBQ}}
receive {reply, ok}
Beispiel:  HBQ ! {<7016.50.0>, {request,dellHBQ,}}
        receive {reply, ok}
```

initHBQ(DLQ-Limit,HBQ-Name): startet den HBQ Prozess auf der HBQ-Node und registriert den Prozess lokal unter dem Namen HBQ-Name. DLQ-Limit gibt die maximale Größe der DLQ an. Initialisiert die HBQ. Als **Rückgabewert** wird die PID des Prozesses gegeben.

{request,pushHBQ,[NNr,Msg,TSclientout]}: fügt eine Nachricht bestehend aus Msg (Textzeile) mit Nummer NNr und dem Sende-Zeitstempel TSclientout (mit `erlang:timestamp()` erstellt) in die HBQ ein. Bei Erfolg wird ein ok als Antwort gesendet. Ist die Nummer NNr kleiner, als die größte Nummer in der DLQ (expectedNr(Queue)), so wird die Nachricht verworfen.

{request,deliverMSG,NNr,ToClient}: beauftragt die HBQ über die DLQ die Nachricht mit der Nummer NNr (falls nicht verfügbar die nächst höhere Nachrichtennummer) an den Client ToClient (als PID) auszuliefern. Bei Erfolg wird die tatsächlich gesendete Nachrichtennummer SendNNr gesendet.

{request,list*ADT*}: Fragt bei der HBQ den aktuellen Inhalt der ADTs HBQ oder DLQ ab. `self()` stellt die Rückrufadresse dar. Als **Rückgabewert** wird in die jeweilige log-Datei der ADT der aktuelle Inhalt geschrieben. Bei Erfolg wird ein ok als Antwort gesendet.

{request,dellHBQ}: terminiert den Prozess der HBQ. Bei Erfolg wird ein ok als Antwort gesendet.

Im Folgenden wird die **Schnittstelle der DLQ** des Servers beschrieben. Die DLQ wird nur von der HBQ aus angesprochen!

```
/* Initialisieren der DLQ */
initDLQ(Size,Datei)

/* Löschen der DLQ */
delDLQ(Queue)

/* Abfrage welche Nachrichtennummer in der DLQ gespeichert werden kann */
expectedNr(Queue)

/* Speichern einer Nachricht in der DLQ */
push2DLQ([NNr,Msg,TSclientout,TShbqin],Queue,Datei)

/* Ausliefern einer Nachricht an einen Leser-Client */
deliverMSG(MSGNr,ClientPID,Queue,Datei)

/* gibt eine Liste der Nachrichtennummern zurück */
listDLQ(Queue)

/* gibt die Größe der DLQ zurück */
lengthDLQ(Queue)
```

initDLQ(Size,Datei): initialisiert die DLQ mit Kapazität Size. Bei Erfolg wird eine leere DLQ zurück geliefert. Datei kann für ein logging genutzt werden.

delDLQ(Queue): löscht die DLQ. Bei Erfolg wird ok zurück geliefert.

expectedNr(Queue): liefert die Nachrichtennummer, die als nächstes in der DLQ gespeichert werden kann. Bei leerer DLQ ist dies 1.

push2DLQ([NNr,Msg,TSclientout,TShbqin],Queue,Datei): speichert die Nachricht [NNr,Msg,TSclientout,TShbqin] in der DLQ Queue und fügt ihr einen Eingangszeitstempel an (einmal an die Nachricht Msg und als expliziten Zeitstempel **TSdlqin** mit erlang:timestamp() an die Liste an. Bei Erfolg wird die modifizierte DLQ zurück geliefert. Datei kann für ein logging genutzt werden. Wenn die vorgegebene Größe der DLQ erreicht wurde, wird beim Eintrag einer neuen Nachricht die älteste Nachricht in der DLQ gelöscht.

deliverMSG(MSGNr,ClientPID,Queue,Datei): sendet die Nachricht MSGNr an den Leser ClientPID. Dabei wird ein Ausgangszeitstempel **TSdlqout** mit erlang:timestamp() an das Ende der Nachrichtenliste angefügt. Sollte die Nachrichtennummer nicht mehr vorhanden sein, wird die nächst größere in der DLQ vorhandene Nachricht gesendet. Bei Erfolg wird die tatsächlich gesendete Nachrichtennummer zurück geliefert. Datei kann für ein logging genutzt werden.

listDLQ(Queue): gibt eine Liste der Nachrichtennummern (ohne Nachricht) zurück. Die Reihenfolge entspricht der Reihenfolge in der DLQ.

lengthDLQ(Queue): gibt die Größe bzw. Länge der DLQ zurück.

Tipp

In der Dateien util.erl gibt es für den Zeitstempel und loggen nützliche Funktionen.

Abnahme

Bis 03. Februar 12:00 Uhr ist die Abgabe der Hausarbeit zu erfolgen. Es gilt der Zeitstempel des E-Mail-Systems der HAW. Die Abgabe muss über Ihren HAW E-Mail account erfolgen.

Die Abgabe besteht aus

1. Die schriftliche Ausarbeitung der Hausarbeit (mit unterschriebener Erklärung zur Eigenständigkeit, siehe dazu die Vorgabe. Die Unterschrift kann per Scan/Foto geleistet werden).
2. Dem von Ihnen entwicklete Code zur Lösung der Aufgabe, der alle Vorgaben erfüllen muss. **In den Sourcedateien ist auf die schriftliche Ausarbeitung zu verweisen**, um die korrekte technische Umsetzung zu dokumentieren.
3. Die geforderten log-Dateien.

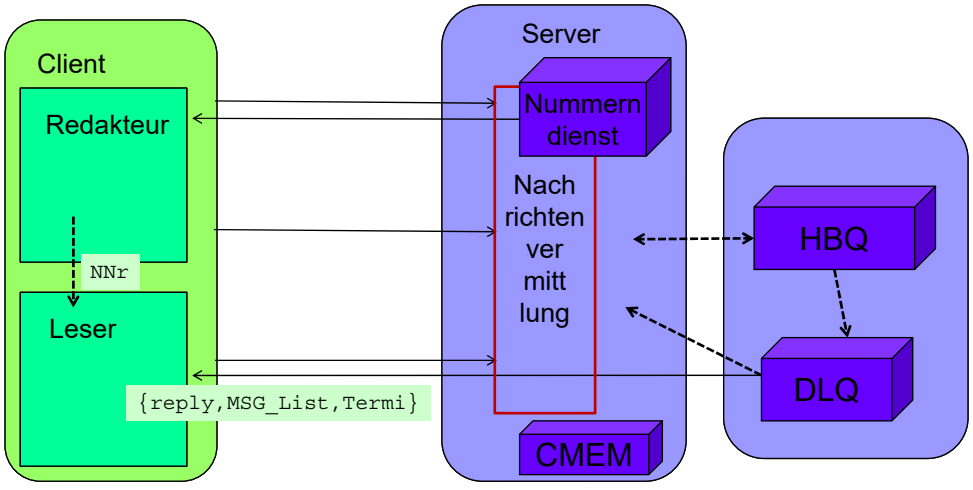
Beachten Sie bitte die Regularien zur Hausarbeit!

Algorithmen und Datenstrukturen

Aufgabe Hausarbeit

1

Nachrichtendienst



MSG_List := [NNr,Msg,TSclientout,TShbqin,TSdlqin,TSdlqout]
[42, "Hallo Server",{1418,898760,161000},{1418,898762,161500},...]

2

Schnittstellen

/* Starten der HBQ */
initHBQ(DLQ-Limit,HBQ-Name)

/* Initialisieren der HBQ */ entfernt auf einer anderen Node
HBQ ! {self(), {request,initHBQ}} receive {reply,ok}

/* Speichern einer Nachricht in der HBQ */ Achtung: zu alte Nachrichten werden verworfen
HBQ ! {self(), {request,pushHBQ,[NNr,Msg,TSclientout]}} receive {reply,ok}

/* Abfrage einer Nachricht */
HBQ ! {self(), {request,deliverMSG,NNr,ToClient}}
receive {reply,SendNNr}

/* Terminierung der HBQ */
HBQ ! {self(), {request,dellHBQ}} receive {reply, ok}

/* Aktueller Stand der ADT in log schreiben */
HBQ ! {self(), {request, listDLQ}} receive {reply, ok}
HBQ ! {self(), {request, listHBQ}} receive {reply, ok}

Schnittstellen

/* Initialisieren der DLQ */
initDLQ(Size,Datei): Integer X Atom → DQueue

/* Abfrage welche Nachrichtennummer in der DLQ gespeichert werden kann */
expectedNr(Queue): DQueue → Integer

/* Speichern einer Nachricht in der DLQ */
/* Bei überschreitung der Größe werden die ältesten Nachrichten gelöscht. */
push2DLQ([NNr,Msg,TSclientout,TShbqin],Queue,Datei) :
MSG_list X DQueue X Atom → DQueue

/* Ausliefern einer Nachricht an einen Leser-Client */
deliverMSG(MSGNr,ClientPID,Queue,Datei):
Integer X PID X DQueue X Atom → Integer

/* Inhalt der DLQ ohne Nachrichten */
listDLQ(Queue): DQueue → Liste

/* Größe der DLQ */
lengthDLQ(Queue): DQueue → Integer

/* Löschen der DLQ */
delDLQ(Queue): DQueue → ok

Schnittstellen

Die HBQ ist komplett in einer Datei hbq.erl zu halten.
Die DLQ ist komplett in einer Datei dlq.erl zu halten.

util.erl kann genutzt werden. Weitere Dateien sind bei den ADTs nicht zulässig (wegen Austauschbarkeit).

„Übergaben“ von komplexeren Datenstrukturen an Erlang OTP ist untersagt: Lediglich die Basis-Strukturen Liste (lists) und Tupel (tuple) dürfen eingesetzt werden (ohne deren komplexen Zugriffsfunktionen!). Algorithmen auf diesen Basisstrukturen müssen selbst implementiert werden.

Mögliche Fragestellungen

- Laufzeit bzw. Durchsatz der ADTs?
- Tests: nur HBQ, nur DLQ, HBQ und DLQ mit gegebenem Client/Server-System
- In welchem Scenario könnte man DLQ/HBQ einsetzen?
- Wie wären die realisierten ADTs einsetzbar?
- Was wären Vorteile von HBQ/DLQ?
- Was wären Nachteile von HBQ/DLQ?

Kein Shared Memory

Message Passing

Prozesse

- Prozesse sind die Grundeinheit der Nebenläufigkeit
- Erlang-Code wird zu Byte-Code kompiliert und durch eine **virtuelle Maschine** ausgeführt, Programme sind somit plattformunabhängig
- Prozesse werden vom Laufzeitsystem (VM) verwaltet, nicht vom BS (d.h. in Kooperation)
- Die Laufzeitumgebung wird als **Node** bezeichnet; einzelne Nodes können miteinander kommunizieren
  - Auf einem Rechner können mehrere Nodes laufen
  - Nodes können über ein Netzwerk verteilt sein

Prozesse

- Verwenden Sie die Standardfunktionen
  - `spawn` zum erzeugen **neuer Prozesse**

```
Pid = spawn(module, function, [arguments]).
```

  - `!` (alias `send`) zum **versenden** von Nachrichten

```
Pid ! message.
```

  und
  - `receive` zum **empfangen** von Nachrichten

```
receive
   message1 -> commands1;
   …
   messagen -> commandsn
end.
```

- Asynchrones senden, synchrones empfangen

Nebenläufigkeits Vorlage

```
-module(template).
-compile(export_all).

start() ->
     spawn(fun() -> loop([]) end).

rpc(Pid, Query) ->
     Pid ! {self(), Query},
     receive {Pid, Reply} -> Reply
     end.

loop(X) ->
 receive {Pid,Query} -> io:format("Rec:~p ! ~p~n",[Pid,Query]),
                        Pid ! {self() , Query},
                        loop(X);
         Any -> io:format("Received:~p~n", [Any]),
                loop(X)
     end.
```

```
werl
0> c(demoRemote).
2> PID = demoRemote:start().
<0.38.0>
3> PID ! da.
Received:da
da
4> demoRemote:rpc(PID,bla).
Received:<0.31.0> ! Bla
ok
5> demoRemote:rpc(PID,bla).
Received:<0.31.0> ! Bla
ok
```


Beispiel: logging

```
% Schreibt auf den Bildschirm und in eine Datei
% nebenläufig zur Beschleunigung
% Beispielaufruf: logging("FileName.log","Textinhalt"),
%
logging(Datei,Inhalt) ->
      spawn( fun() -> io:format(Inhalt),
                      file:write_file(Datei,Inhalt,[append])
                      end).
```

Links und Trapping-Exits

- Fehlerketten werden über gebundene Prozesse definiert
- Wenn ein Prozess beendet wird, sendet er ein exit-Signal an seine gebundenen Prozesse
- `spawn_link` zum erzeugen **gebundener neuer Prozesse**

```
Pid = spawn_link(module, function, [arguments]).
```

```
start() ->
  process_flag(trap_exit, true),
  Pid = spawn_link(Mod, Fun, Args),
  receive  {'EXIT', Pid, Why} ->
                  handle_process_crash(Why,...);
           ...
  end.
```


Tabellen und Datenbanken

- **Wörterbuch** (Term Storage) von Erlang
  - `ets` Tabellen sind RAM-basierte (transient)
  - `dets` (`dISK-ets`) Tabellen sind auf der Festplatte (persistent)
- `Mnesia` ist eine verteilte Real-time **Datenbank** in Erlang
  - Query Sprache sieht aus wie SQL/List Comprehensions
  - Enthält Standard-Visualisierungstools
- und weitere Bibliotheken…

Beispiel: System starten

Starten des Servers:

(w)erl -(s)name wk -setcookie zummsel

1> server_starter:start( ).

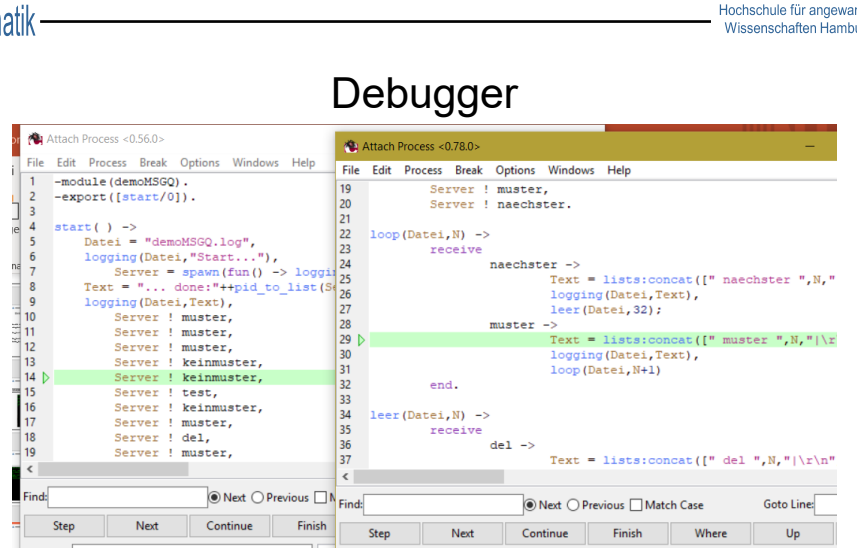
% in der **server.cfg**:

% {lifetime, 60}. Zeit in Sekunden, die der Server auf Aufträge wartet

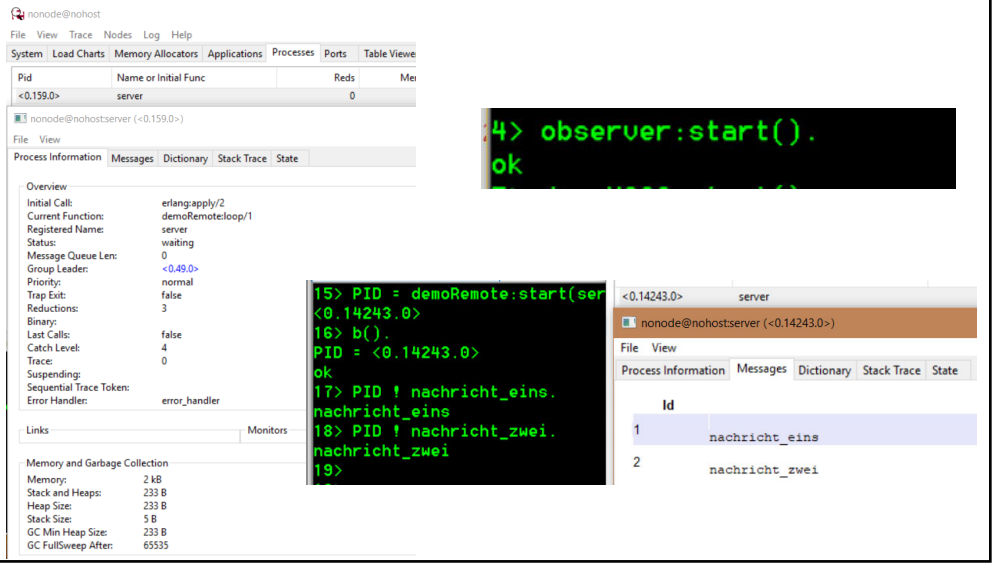
% in der **server_starter.erl**

```
%-module(server_starter).
%-export([start/0]).
%start() -> {ok, ConfigListe} = file:consult("server.cfg"),
%          {ok, Lifetime} = werkzeug:get_config_value(lifetime, ConfigListe),
%          {ok, HostName} = inet:gethostname(),
%          Datei = lists:concat(["NServer@",HostName,".log"]),
%          ServerPid = spawn(fun() -server_func:start(Datei,Lifetime) end),
%          register(Servername,ServerPid).
```


Debugger



observer



process_info

```
13> process_info(PID).
[{registered_name,server},
 {current_function,{demoRemote,loop,1}},
 {initial_call,{erlang,apply,2}},
 {status,waiting},
 {message_queue_len,2},
 {messages,[nachricht_1,nachricht_2]},
 {links,[]},
 {dictionary,[]},
 {trap_exit,false},
 {error_handler,error_handler},
 {priority,normal},
 {group_leader,<0.49.0>},
 {total_heap_size,233},
 {heap_size,233},
 {stack_size,5},
 {reductions,5},
 {garbage_collection,[{max_heap_size,#{error_logger => true,
                                       kill => true,
                                       size => 0}},
                      {min_bin_vheap_size,46422},
                      {min_heap_size,233},
                      {fullsweep_after,65535},
                      {minor_gcs,0}]},
 {suspending,[]}]
14>
```