# CAV – Tool Dokumentation

von

Christian Sternagel

# Inhaltsverzeichnis


**Einleitung: Was ist Stauvermeidung** 3
AIMD .......... 3
**Die Bedienoberfläche des CAV – Tools** 5
Das CAV – Tool .......... 5
Globale Einstellungen für Simulationen .......... 5
Eingaben für die beiden Sender synchronisieren .......... 5
Dauer einer Simulation .......... 6
Die Sendegeschwindigkeit .......... 6
Schaltfläche für Sender 1 .......... 6
Fehlermeldungen .......... 7
Die Ausgabefläche .......... 7
Schaltfläche für Sender 2 .......... 7
Logging und andere Einstellungen .......... 8
Logdateien erzeugen .......... 8
Weitere Kurven anzeigen .......... 8
Drei Buttons .......... 8
Repaint .......... 8
Clear .......... 9
Exit .......... 9
**Ein erläuterndes Beispiel: AIMD vs. MIMD** 10
**Interna** 14
Klassendiagramm .......... 14
Überblick über die wichtigsten Klassen .......... 15
CAV_Filter .......... 15
C_System .......... 15
CAV_Tool .......... 15
InputSynchronizer .......... 15
InputSynchronizerConstants .......... 16
Param .......... 16
ParamSlider .......... 16
Screen .......... 16
ScriptParser .......... 16
Sender .......... 16
SenderUI .......... 17
Tokenizer .......... 17
Tokens .......... 17
Die verwendeten Grammatiken .......... 17
Die Grammatik des `ScriptParser` .......... 17
Die Grammatik des `Tokenizer` .......... 18
**Anhang** 19
1 Introduction .......... 19
2 Description .......... 19
2.1 A Simple Example .......... 19
2.2 Preprocessor-Directives .......... 20
2.3 Function Calls .......... 21
2.4 Comments .......... 22
**Literatur** 23

# Einleitung:
# Was ist Stauvermeidung (*Congestion Avoidance*)?

Stauvermeidungsverfahren erlauben einem (Computer -) Netzwerk (wie z.B. dem Internet) Verzögerungen und Paketverluste so gering wie möglich zu halten, aber andererseits einen möglichst hohen Durchsatz zu erzielen. Gleichzeitig sollen dabei alle Beteiligten (im Folgenden Sender genannt) möglichst „fair“ behandelt werden, was so viel bedeutet, als dass jedem Sender in etwa dieselbe Senderate zur Verfügung stehen sollte.

Nun gibt es verschiedenste Verfahren dies zu erreichen. Wohl das verbreitetste ist AIMD („Additive Increase Multiplicative Decrease“), welches in ähnlicher Form wie unten beschrieben auch von TCP/IP verwendet wird.

## AIMD

Wie der Name schon sagt, funktioniert AIMD, indem die Senderate eines Senders um einen konstanten Wert erhöht wird (additive) bzw. um einen konstanten Faktor reduziert wird (multiplicative). Um zu entscheiden, ob der Sender zu einem bestimmten Zeitpunkt die Senderate erhöhen oder reduzieren soll, benötigt er eine Rückmeldung („Feedback“) vom Empfänger, denn dieser kann feststellen, ob Pakete verloren gehen, die Senderate also zu hoch ist oder das Netzwerk nicht ganz ausgelastet ist, also die durch das Netzwerk zur Verfügung stehende Datentransferrate durch die Summe aller Senderaten aller Sender nicht erreicht wird. Diese Art von Feedback nennt man „binäres Feedback“ („binary feedback“), da es nur zwei mögliche Rückmeldungen gibt.

*Ein Beispiel:*

In diesem Beispiel gehen wir von folgendem, vereinfachten Modell aus: Es gibt zwei Sender *S1, S2*. Die Datentransferrate des Netzwerks wird über die Variable *traffic* gemessen (dabei entspricht *traffic* = 1.0 100% Netzauslastung). Die Senderaten der Sender werden über die Variablen *rate1* und *rate2* gemessen. Der Wert von *traffic* zu einem bestimmten Zeitpunkt ergibt sich aus der Summe von *rate1* und *rate2*. Sei *beta* = 0.5 der Faktor, um den die Senderaten verringert werden bzw. *alpha* = 0.1 der Wert, um den diese erhöht werden. Weiters sollen die Sender synchron senden. Wird nun AIMD verwendet, wird jedes Mal, wenn *traffic* größer als eins ist, sowohl *rate1* als auch *rate2* mit *beta* multipliziert (also auf die Hälfte reduziert) und solange *traffic* kleiner gleich eins ist, *rate1* und *rate2* jeweils um 0.1 erhöht. Hier das ganze als Pseudocode aus Sicht eines einzelnen Senders:

```
Falls traffic > 1
        rate = rate * beta
sonst
        rate = rate + alpha
```

Jeder Sender entscheidet also auf Grund der Rückmeldung des Empfängers, ob er seine Senderate erhöht oder reduziert.

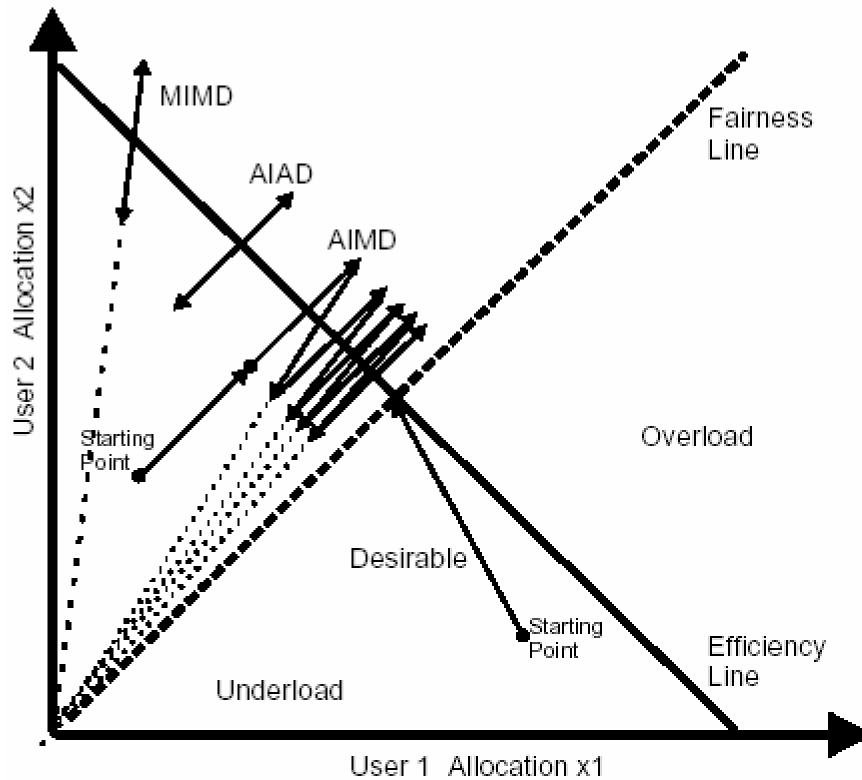


*Abbildung 1: Stauvermeidungsverfahren*

In *Abbildung 1* sind mehrere Stauvermeidungsverfahren dargestellt. Es wird der Fall von zwei Benutzern behandelt, die synchron senden.(Also eine stark vereinfachtes Modell). *User1* und *User2* entsprechen dabei *S1* und *S2* aus unserem Beispiel. Diese Art von Vektordiagramm wird in [1] detailliert beschrieben (*siehe Seite 7*). Die gestrichelte Linie (diagonal) stellt die „Fairness Line“ dar d.h. wenn ein Punkt auf dieser Linie liegt, haben beide Sender dieselbe Bandbreite zur Verfügung (Ziel eines Stauvermeidungsverfahrens sollte es sein, von jedem Startpunkt möglichst schnell auf die „Fairness Line“ zu kommen und dann auch dort zu bleiben). Die durchgehende Linie (diagonal) stellt die „Efficiency Line“ dar. Je näher ein Punkt an diese Linie kommt, desto effizienter wird die zur Verfügung stehende Bandbreite benutzt. Liegt ein Punkt rechts von dieser Linie, kommt es zu einer Überlastung und Datenpakete gehen verloren. Liegt ein Punkt links von dieser Linie, werden die Ressourcen nicht ausgenutzt (Am effizientesten und „fairsten“ ist also der Kreuzungspunkt der „Fairness Line“ mit der „Efficiency Line“).

Man sieht, dass im Gegensatz zu AIAD („Additive Increase Additive Decrease“) und MIMD („Multiplicative Increase Multiplicative Decrease“) bei AIMD gleichzeitig mit der Stauvermeidung auch eine Annäherung an einen „fairen“ Punkt stattfindet (das Optimum entspricht dem Kreuzungspunkt der „Fairness Line“ mit der „Efficiency Line“).

## Die Bedienoberfläche des CAV – Tool

Es handelt sich beim CAV – Tool um ein simples, aber dennoch mächtiges Tool zur Visualisierung des Verhaltens verschiedener Stauvermeidungsverfahren. Im Großen und Ganzen ein Simulator der auf die Vektordiagramme im Sinne von Chiu/Jain aufbaut, stellt das CAV – Tool sowohl Forschern, als auch Lehrern und Studenten eine neue Abstraktionsstufe zur Verfügung, die bereits in der Praxis äußerst nützlich war.

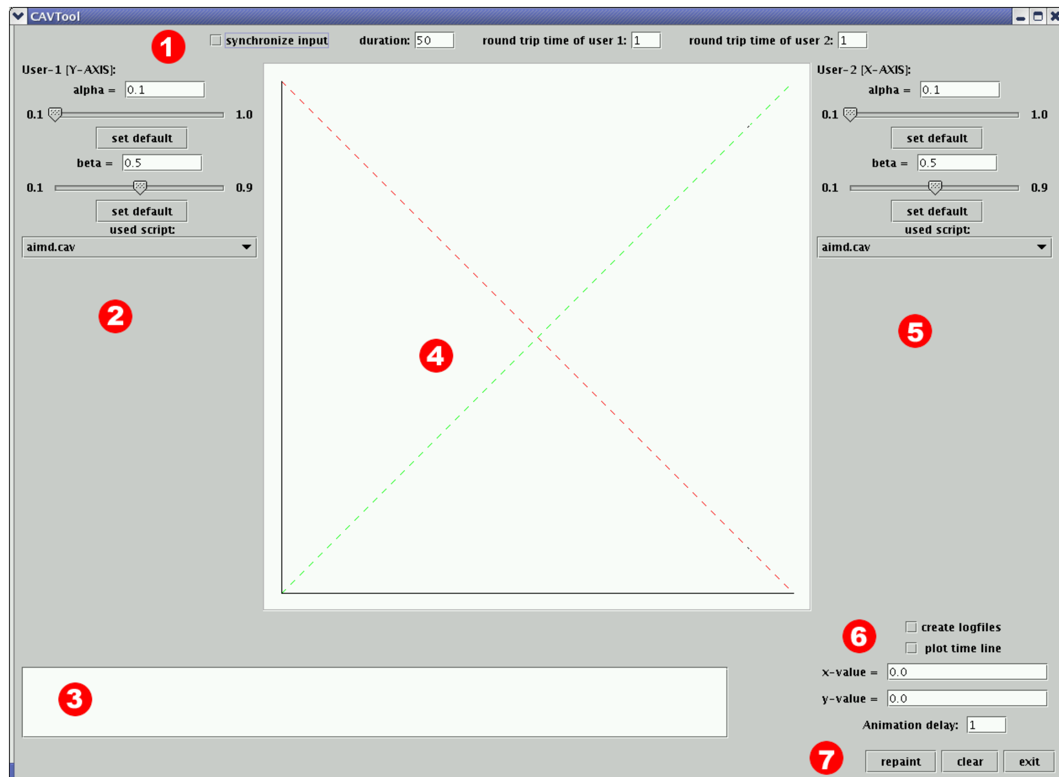


*Abbildung 2: Die Bedienoberfläche des CAV – Tool*

## Das CAV – Tool

Im Folgenden wird die Bedienung des CAV – Tool beschrieben. Zu diesem Zweck werden die einzelnen Sektionen, wie in *Abbildung 2* dargestellt, der Reihe nach besprochen.

### Globale Einstellungen für Simulationen

Siehe *Abbildung 2* Nummer 1.

Es gibt vier globale Einstellungen, mit deren Hilfe Simulationsparameter gesetzt werden können. Die hier gesetzten Einstellungen bleiben so lange wirksam, bis sie vom Benutzer wieder geändert werden, wirken sich also auf alle folgenden Simulationen aus.

### Eingaben für die beiden Sender synchronisieren

Wird die Option *synchronize input* verwendet, wirken sich alle Änderungen, die bei Sender 1 vorgenommen werden (verwendete Scriptdatei, Werte für Parameter, etc.), gleichermaßen auf Sender 2 (siehe *Seite 7*) aus. Dabei ist zu beachten, dass eine manuelle Änderung (gemeint ist eine Änderung über die Tastatur) des Werts eines Parameters (siehe *Seite 6* bzw. *Seite 7*) erst nach Betätigung der `<ENTER>` Taste gültig wird.

### Dauer einer Simulation

Über das Textfeld *duration* wird bestimmt, wie viele Zeiteinheiten eine Simulation laufen soll. Der Defaultwert ist dabei 50. Die tatsächliche Dauer einer Zeiteinheit hängt dabei von der auf *Seite 8* besprochenen *Animation delay* ab. Die Defaulteinstellung ist hier eine Millisekunde.

### Die Sendegeschwindigkeit

Die beiden Textfelder *round trip time of user 1* und *round trip time of user 2* geben an, wie häufig der jeweilige Sender eine Nachricht sendet und somit Ressourcen für sich beansprucht. Gilt für beide Sender derselbe Wert, so handelt es sich um eine Simulation für den synchronen Fall, andernfalls um eine Simulation für den asynchronen Fall. Ein Wert von zwei würde z.B. bedeuten, dass der Sender nur alle zwei Zeiteinheiten sendet. Der Defaultwert ist für beide Sender eins (also der synchrone Fall, dass beide Sender jede Zeiteinheit senden).

## Schaltfläche für Sender 1

Siehe *Abbildung 2* Nummer 2.

Für jeden Parameter, den ein Stauvermeidungsverfahren (Parameter werden über eine Script – Datei definiert; näheres hierzu siehe *Anhang*.) besitzt, gibt es hier drei Bedienelemente:

- Ein Textfeld, in dem der aktuelle Wert des Parameters (der Name des Parameters steht links davon) steht. Über dieses Textfeld kann der Wert auch verändert werden (Hinweis: nach Eingabe über die Tastatur ist das Betätigen der `<ENTER>` Taste nötig, um die Änderung wirksam zu machen).
- Ein Schieberegler mit dessen Hilfe der Wert des Parameters in Schritten von 0.1 Einheiten verändert werden kann. Soll die Abstufung differenzierter sein, muss der exakte Wert über das oben genannte Textfeld eingegeben werden. (Hinweis: das Dezimaltrennzeichen ist der Punkt, nicht das Komma)
- Ein Button *set default*, über den der Parameter auf seinen Defaultwert gesetzt werden kann.

Ganz unten in dieser Schaltfläche kann unter *used script* die Scriptdatei bestimmt werden, die der Sender verwenden soll. Zur Auswahl stehen alle *.cav Dateien, die sich im aktuellen Verzeichnis befinden (wurde das CAV – Tool z.B. von ~/CAVTool aus gestartet, ist dies auch das aktuelle Verzeichnis).

## Fehlermeldungen

Siehe *Abbildung 2* Nummer 3.

In diesem Textfeld werden alle evt. Fehlermeldungen ausgegeben, die während einer Session mit dem CAV – Tool anfallen. Diese Meldungen beziehen sich immer auf eine Scriptdatei. Hat man zum Beispiel in einer Scriptdatei namens *myscript.cav* in der ersten Zeile einen Parameter *alpha* definiert, dessen Wertebereich zwischen 0.1 und 0.9 liegt (siehe Anhang für detailliertere Informationen zu Scriptdateien) und will nun eine Simulation laufen lassen, in der man diesem Parameter einen Wert von 1.0 zugeordnet hat, erhält man folgende Fehlermeldung:

```
myscript.cav at line 0: 1.0 is not within the range of alpha ...
```

(Hinweis: Als Zeilennummer wird deshalb 0 angegeben, weil der Fehler schon beim Lauf des Präprozessors aufgetreten ist und daher der Interpreter noch gar nicht begonnen hat die Scriptdatei zu lesen. Präprozessordirektiven müssen in einer Scriptdatei immer ganz am Anfang stehen)

## Die Ausgabefläche

Siehe *Abbildung 2* Nummer 4.

Hier wird das Vektordiagramm, das bei einer Simulation entsteht, geplottet. Auf der y - Achse (vertikal) wird dabei aufgetragen, wie viel Prozent der zur Verfügung stehenden Datentransferrate Sender 1 zu einem bestimmten Zeitpunkt verwendet. Auf der x - Achse (horizontal) wird die Ressourcenbelegung zu einem bestimmten Zeitpunkt von Sender 2 aufgetragen.

## Schaltfläche für Sender 2

Siehe *Abbildung 2* Nummer 5.

Wie für Sender 1 besprochen.

## Logging und andere Einstellungen

Siehe *Abbildung 2* Nummer 6.

Mit Hilfe der *Animation delay* kann bestimmt werden, wie lange (in Millisekunden) zwischen dem Zeichnen von zwei Punkten gewartet werden soll. Dadurch ist es möglich die zeitliche Entwicklung besser zu verfolgen.

### Logdateien erzeugen

Durch Auswahl der Option *create logfiles* wird das Programm angewiesen, für jede Simulation mehrere Logdateien zu erzeugen. Eine Logdatei besteht dabei aus einer einfachen Textdatei, in der zweispaltig Zahlen stehen. Dabei repräsentiert die erste Spalte alle x - Werte und die zweite alle y - Werte. Es gibt Programme wie z.B. `gnuplot` unter UNIX, die solche Dateien zur Ausgabe benutzen.

Die vier Logdateien die entstehen sind folgende:

- *distance-from-ideal.log*: Zeigt den Abstand vom Optimum über die Zeit an (das Optimum ist der Kreuzungspunkt der „Efficiency Line“ mit der „Fairness Line“).
- *first-user-rate-time.log*: Zeigt die benutzte Bandbreite von Sender 1 über die Zeit an.
- *second-user-rate-time.log*: Zeigt die benutzte Bandbreite von Sender 2 über die Zeit an.
- *two-user-trajectory.log*: Trägt die benutzten Bandbreiten beider Sender gegeneinander auf (sowie in der Ausgabefläche; siehe Seite XX).

Ein Beispiel für `gnuplot`:

```
$] gnuplot
gnuplot> plot "two-user-trajectory.log" with linespoints
```

### Weitere Kurven anzeigen

Durch Auswahl der Option *plot timeline* wird in einem neuen Fenster eine Übersicht ausgegeben. Hier werden drei Kurven geplottet. Die rote Kurve zeigt die Senderate von Sender 1 über die Zeit, die grüne Kurve dieselbe Information für Sender 2 und die blaue Kurve schließlich zeigt den Abstand vom Optimum zu einem bestimmten Zeitpunkt.

## Drei Buttons

Siehe *Abbildung 2* Nummer 7.

### Repaint

Mit Hilfe des `repaint` Buttons wird eine Kurve gezeichnet, die alle aktuellen Einstellungen verwendet und deren Startpunkt den in *x-value* = und *y-value* = gespeicherten Koordinaten entsprechen. Dadurch kann man z.B. verschiedene Methoden mit demselben Startpunkt vergleichen oder exakt einen Startpunkt bestimmen. (Hinweis: Normalerweise wird der Startpunkt durch einen Mausklick in die Ausgabefläche bestimmt.)

**Clear**

Durch Betätigung von `clear` wird sowohl der Inhalt des Ausgabefensters als auch alle Fehlermeldungen gelöscht.
(Hinweis: Durch drücken der rechten Maustaste kann das Ausgabefenster gelöscht werden, ohne die Fehlermeldungen ebenfalls zu löschen.)

**Exit**

Durch drücken des `exit` Buttons wird das CAV – Tool beendet. Alle zugehörigen Fenster werden geschlossen.

## Ein erläuterndes Beispiel: AIMD vs. MIMD

Es folgt ein ausführliches Beispiel in dem zwei Stauvermeidungsverfahren verglichen werden. Als erstes die Script – Datei zu AIMD:

```
01:  #define_param alpha range 0.1 to 1 default 0.1
02:  #define_param beta  range 0.1 to 1 default 0.5
03:
04:  maxrate = 1.0;        # nicht mehr als 100% Auslastung
05:
06:  if(traffic > 1){      # Staugefahr!
07:       rate = rate * beta;
08:  }
09:  else{                 # eine höhere Rate ist möglich
10:       rate = rate + alpha;
11:       rate = min(rate, maxrate);
12:  }
```

Dann die Script – Datei zu MIMD:

```
01:  #define_param alpha range 1 to 2 default 1.2
02:  #define_param beta  range 0.1 to 1 default 0.5
03:
04:  maxrate = 1.0;        # nicht mehr als 100% Auslastung
05:
06:  if(traffic > 1){      # Staugefahr!
07:       rate = rate * beta;
08:  }
09:  else{                 # eine höhere Rate ist möglich
10:       rate = rate * alpha;
11:       rate = min(rate, maxrate);
12:  }
```

Wie man sieht unterscheiden sich die beiden Dateien nicht sehr. Einzig und allein die Tatsache, dass bei MIMD auch die Erhöhung der Senderate durch einen multiplikativen Faktor stattfindet, unterscheidet die beiden Strategien. Nun wollen wir sehen, was dabei herauskommt, wenn man das CAV – Tool mit demselben Startwert (0.3 für Sender 1 und 0.5 für Sender 2) für jede Script – Datei einmal laufen lässt. Dabei wird *duration* auf 10 gesetzt, was bedeutet, dass 10 Punkte (exklusive Startpunkt) ausgegeben werden.

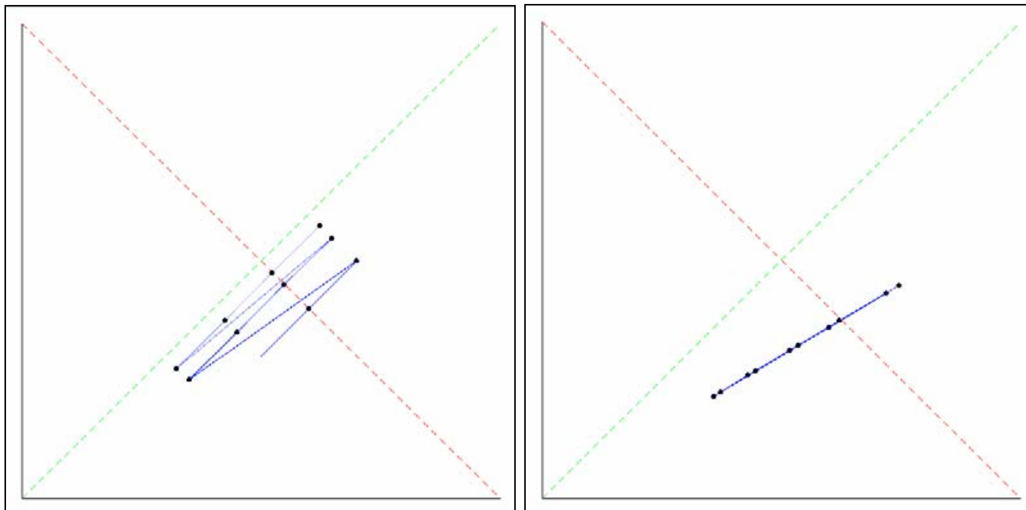

*Abbildung 3: links AIMD, rechts MIMD*

Wie man in *Abbildung 3* sieht, haben die Punkte bei AIMD eine Tendenz, sich der „Fairnes Line“ zu nähern, während sich bei MIMD alle Punkte auf der selben Linie befinden und diese nie verlassen.

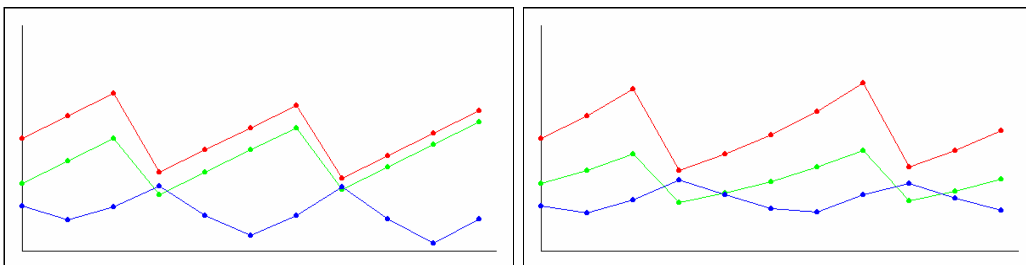

*Abbildung 4: links AIMD, rechts MIMD*
*rot ... Senderate von Sender 1*
*grün ... Senderate von Sender 2*
*blau ... Abstand vom Optimum*

*Abbildung 4* zeigt für beide Verfahren die Kurven, die bei Auswahl der Option *plot timeline* geplottet werden. Hier ist leicht zu erkennen, dass sich AIMD „fairer“ verhält als MIMD, da die Kurven von Sender 1 und Sender 2 bei AIMD näher zusammenliegen. Außerdem ist zu erkennen, dass sich der Abstand zum Optimum global gesehen bei AIMD verringert, bei MIMD jedoch stets gleich bleibt.

Bei Auswahl der Option *create logfiles* entstehen bei AIMD folgende vier Dateien:

`distance-from-ideal.log:`

```
0      0.2
1      0.14142135623730948
2      0.19999999999999996
3      0.29154759474226505
4      0.158113883008419
5      0.07071067811865474
6      0.15811388300841886
7      0.28504385627478457
8      0.14577379737113255
9      0.03535533905932733
10     0.14577379737113239
```

`first-user-rate-time.log:`

```
0      0.3
1      0.4
2      0.5
3      0.25
4      0.35
5      0.44999999999999996
6      0.5499999999999999
7      0.27499999999999997
8      0.375
9      0.475
10     0.575
```

`second-user-rate-time.log:`

```
0      0.5
1      0.6
2      0.7
3      0.35
4      0.44999999999999996
5      0.5499999999999999
6      0.6499999999999999
7      0.32499999999999996
8      0.42499999999999993
9      0.5249999999999999
10     0.6249999999999999
```

`two-user-trajectory.log:`

```
0.5                   0.3
0.6                   0.4
0.7                   0.5
0.35                  0.25
0.44999999999999996   0.35
0.5499999999999999    0.44999999999999996
0.6499999999999999    0.5499999999999999
0.32499999999999996   0.27499999999999997
0.42499999999999993   0.375
0.5249999999999999    0.475
0.6249999999999999    0.575
```

Und bei MIMD diese hier:

```
distance-from-ideal.log:
      0       0.2
      1       0.17204650534085253
      2       0.23026940743398805
      3       0.31663227883461287
      4       0.2502171856607775
      5       0.18985373738749525
      6       0.17598180560501134
      7       0.2519423309048322
      8       0.3037603241462585
      9       0.23705803870916003
      10      0.18142672943078808
```

```
first-user-rate-time.log:
      0       0.3
      1       0.36
      2       0.432
      3       0.216
      4       0.2592
      5       0.31104
      6       0.37324799999999997
      7       0.44789759999999995
      8       0.22394879999999998
      9       0.26873855999999996
      10      0.32248627199999996
```

```
second-user-rate-time.log:
      0       0.5
      1       0.6
      2       0.72
      3       0.36
      4       0.432
      5       0.5184
      6       0.62208
      7       0.7464959999999999
      8       0.37324799999999997
      9       0.44789759999999995
      10      0.5374771199999999
```

```
two-user-trajectory.log:
      0.5                   0.3
      0.6                   0.36
      0.72                  0.432
      0.36                  0.216
      0.432                 0.2592
      0.5184                0.31104
      0.62208               0.37324799999999997
      0.7464959999999999    0.44789759999999995
      0.37324799999999997   0.22394879999999998
      0.44789759999999995   0.26873855999999996
      0.5374771199999999    0.32248627199999996
```

Alle Dateien sind Textdateien und haben das selbe Format. In der linken Spalte stehen alle x – Werte und in der rechten alle y – Werte (*siehe Anhang*).

# Interna

Der folgende Abschnitt widmet sich dem internen Aufbau des CAV – Tool, wie z.B. die Klassenstruktur der Java – Programmierung oder die verwendete Grammatik der Scriptsprache.

## Klassendiagramm

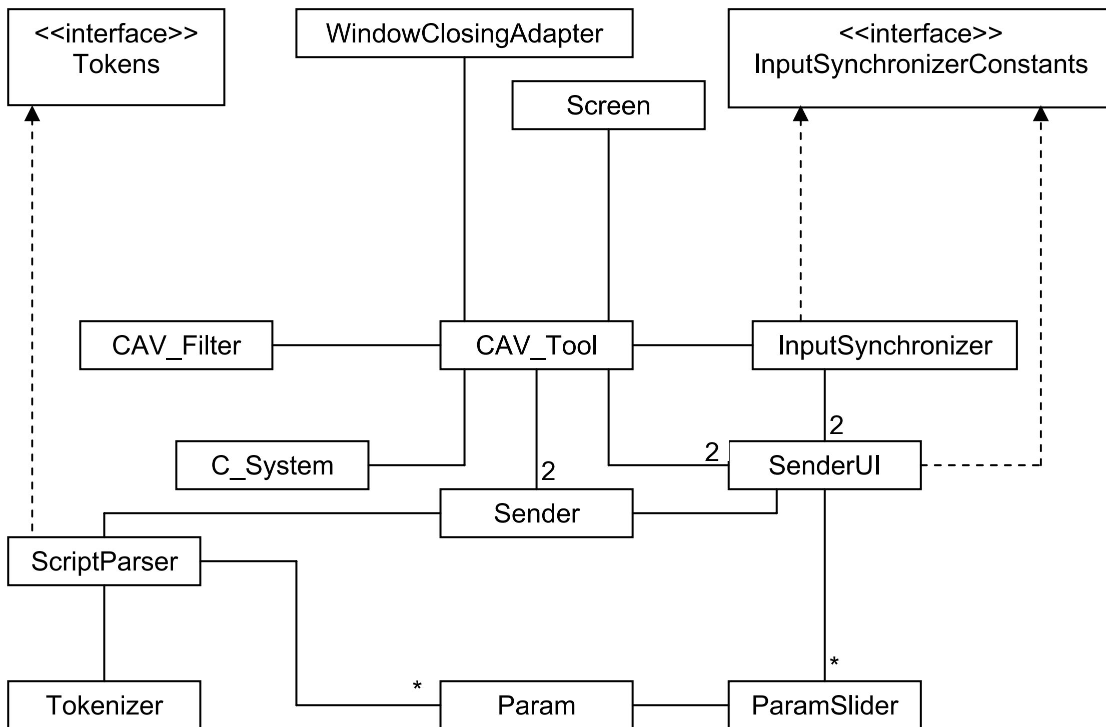


*Abbildung 5: Die Klassenstruktur des CAV – Tool*

In obiger Abbildung ist die Klassenstruktur des CAV – Tool dargestellt. Interfaces sind als solche markiert und Vererbungsbeziehungen (durch das Java – Schlüsselwort `implements`) sind durch gestrichelte Linien dargestellt, die in Richtung des Interfaces eine volle Pfeilspitze besitzen. Z.B. implementiert die Klasse `ScriptParser` das Interface `Tokens`. Die Multiplizität von Assoziationen (durch Striche verbundene Klassen) wird durch Bezeichner (2, *) angegeben oder implizit (durch Fehlen eines Bezeichners) als einfach gekennzeichnet. Dabei bedeutet ein “*“, dass beliebig viele (also auch keine) Instanzen der durch “*“ gekennzeichneten Klasse mit einer anderen Klasse assoziiert sein können.

In *Abbildung 5* werden nur Klassen gezeigt, die noch nicht im `JRE` (Java Runtime Environment) implementiert sind. Auch Vererbungsbeziehungen mit schon vorhandenen Klassen des `JRE` wurden weggelassen.

## Überblick über die wichtigsten Klassen

Es folgt eine Erläuterung der wichtigsten Klassen des CAV – Tool. Programmiertechnische Einzelheiten können dem Sourcecode bzw. der Klassendokumentation im `javadoc` – Format (HTML) entnommen werden.

### CAV_Filter

Diese Klasse sorgt dafür, dass alle *.cav Dateien im Arbeitsverzeichnis (das Verzeichnis aus dem das CAV – Tool gestartet wurde) in der Benutzeroberfläche als Script – Dateien zur Verfügung stehen. Besitzt eine Datei also nicht die Endung .cav, wird sie auch nicht als Script – Datei erkannt. (Hinweis: Unter Windows ist es in einigen Texteditoren nur dann möglich eine andere Endung als .txt zu verwenden, wenn der Dateiname in Anführungsstrichen geschrieben wird. Natürlich ist es immer möglich, eine Datei im Nachhinein umzubenennen.)

### C_System

Der Name dieser Klasse steht für „Canvas System“, was schon einen Hinweis auf die Verwendung gibt. `C_System` ist dafür konzipiert, drei verschiedenfärbige (rot, grün, blau) Kurven darzustellen. Dabei ist es möglich, nach und nach Punkte ans Ende einer Kurve „anzuhängen“. Die x – Achse skaliert in Abhängigkeit der Anzahl der Punkte, diese wiederum hängt von der Dauer der Simulation ab (siehe *Seite 6)*.

### CAV_Tool

Hierbei handelt es sich um die zentrale Klasse des CAV – Tool. Von hier wird die Applikation gestartet und über diese Klasse sind alle anderen Klassen verknüpft. Gleichzeitig beinhaltet `CAV_Tool` die graphischen Elemente des CAV – Tool und verwaltet diese.

### InputSynchronizer

Falls die Option `synchronize input` (siehe *Seite 5*) verwendet wird, sorgt diese Klasse dafür, dass alle Veränderungen bei einem der Sender sich auch auf den anderen auswirken. Das verwendete Design – Pattern entspricht im Großen und Ganzen dem *Observer – Pattern* [3]. Der `InputSynchronizer` des `CAV_Tool` registriert sich bei beiden `SenderUI`s. Diese wiederum benachrichtigen den `InputSynchronizer` bei jeder Änderung des eigenen Zustands.

### InputSynchronizerConstants

Diese Klasse definiert einige Konstanten, um die genaue Art der Änderung eines `SenderUI` zu bestimmen (wurde ein Schieberegler betätigt, wurde eine andere Skript – Datei ausgewählt etc.).

### Param

Wie im Anhang erläutert (siehe *Seite 20*), ist es möglich mit Hilfe von Präprozessordirektiven, veränderliche Parameter zu definieren. Für jeden solchen Parameter wird in jedem `ScriptParser` eine Instanz von `Param` erzeugt, um den augenblicklichen Wert und den Wertebereich („Domain“) eines Parameters zu speichern.

### ParamSlider

In der Benutzeroberfläche existiert für jede Instanz von `Param` ein `ParamSlider` (maximal 10 pro `SenderUI`) der dazu dient, die Werte eines Parameters zu verändern (entweder mit Hilfe der Maus über einen Schieberegler oder durch Eingabe eines exakten Werts über die Tastatur in einem speziell dafür vorgesehenen Textfeld, das auch von `ParamSlider` verwaltet wird).

### Screen

Hierbei handelt es sich um die Ausgabefläche des CAV – Tool (siehe *Seite 7*).

### ScriptParser

Für jeden der beiden `SenderUI`s existiert ein eigener `ScriptParser`, der jedes Mal, wenn ein `Sender` sendet die zugehörige Script – Datei interpretiert und auf Grund des in dieser Datei implementierten Stauvermeidungsverfahrens (unter Berücksichtigung aller Parameter) die neue Senderate bestimmt. Die von `ScriptParser` verwendete Grammatik ist weiter unten zu finden.

### Sender

Es gibt zwei `Sender`, die jeweils einen eigenen `ScriptParser` besitzen. Diese Klasse stellt die Brücke zwischen der graphische Klasse `SenderUI` und der Klasse `ScriptParser` dar. Jegliche Änderung eines `SenderUI` wird vom zugehörigen `Sender` an den zugehörigen `ScriptParser` delegiert (*Delegate – Pattern* siehe [3]).

### SenderUI

Alle graphischen Bedienelemente, die dazu benötigt werden, die Parameter eines Senders zu verändern werden von `SenderUI` verwaltet. `SenderUI` benachrichtigt bei jeder Änderung `InputSynchronizer`, dieser kontrolliert, ob eine Synchronisation erwünscht ist (Option *synchronize input;* siehe *Seite 5*) und führt diese, falls erwünscht, aus.

### Tokenizer

`Tokenizer` zerlegt den Inhalt einer Script – Datei in Tokens, wie sie von `ScriptParser` erwartet werden. Die dabei verwendete Grammatik wird weiter unten beschrieben (siehe *Seite 17*).

### Tokens

Das Interface `Tokens` definiert alle Konstanten, die von `ScriptParser` benötigt werden (z.B. ob es sich bei einem bestimmten Token um eine Gleitkommazahl, einen Operator etc. handelt).

## Die verwendeten Grammatiken

Im Folgenden wird sowohl die Grammatik des Interpreters (`ScriptParser`) als auch die des Tokenizers (`Tokenizer`) besprochen. Die dabei verwendete Darstellungsform entspricht der BNF – Notation.

### Die Grammatik des `ScriptParser`

```
<script> ::= END | <statement> END

<statement> ::= <expression>; | <compound> | <selection> | <statement> <statement>

<compound> ::= { <statement-sequence> }

<selection> ::= if<expression><compound>else<compound> | if<expression><compound>

<expression> ::= <and-expression> | <expression> || <and-expression>

<and-expression> ::= <equ-expression> | <and-expression> && <equ-expression>

<equ-expression> ::= <rel-expression> | <equ-expression> == <rel-expression>
        | <equ-expression> != <rel-expression>

<rel-expression> ::= <add-expression> | <rel-expression> < <add-expression>
        | <rel-expression> > <add-expression>
        | <rel-expression> <= <add-expression>
        | <rel-expression> >= <add-expression>

<add-expression> ::= <add-expression> + <term> | <add-expression> - <term>
        | <term>

<term> ::= <term> / <primary> | <term> * <primary> | <primary>

<primary> ::= NUMBER | NAME | NAME = <expression> | -<primary> | ( <expression> )
```

Alle Terminalsymbole sind **fett** gedruckt. Beim Terminalsymbol **NUMBER** handelt es sich um eine beliebige Zahl (beliebig soll bedeuten, dass nicht zwischen Gleitkommazahlen und Ganzzahlen unterschieden wird), **NAME** steht für einen Variablenbezeichner und **END** schließlich steht für das Dateiende. Alle Operatoren (ebenfalls Terminalsymbole) verhalten sich wie von C++ gewohnt, außerdem ist die obige Grammatik eine vereinfachte Form der in [4] in Anhang A beschriebenen. Das bedeutet auch, dass alle Vorrangregeln denen von C++ entsprechen (soweit überhaupt anwendbar).

### Die Grammatik des `Tokenizer`

```
<token> ::= <identifier> | <number> | <operator> | <literal>

<identifier> ::= <letter> | <letter><alnum>

<alnum> ::= <letter> | <digit> | <alnum><alnum>

<letter> ::= _ | a | ... | z | A | ... | Z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<digit-seq> ::= <digit> | <digit><digit-seq> | null

<number> ::= <digit-seq>.<digit-seq> | <digit-seq>.<digit-seq>e<digit-seq>
        | <digit-seq>.<digit-seq>e-<digit-seq>
        | <digit-seq>.<digit-seq>E<digit-seq>
        | <digit-seq>.<digit-seq>E-<digit-seq>

<operator> ::= * | / | + | - | % | = | < | > | ! | & | <operator><operator> | |

<literal> ::= any other character
```

Alle Terminalsymbole sind wieder **fett** gedruckt. **null** bedeutet, dass gar nichts hier steht (also ist ".0e" eine gültige Zahl). Mit **letter** ist jeder Buchstabe des Alphabets oder der Unterstrich gemeint. Auf Grund obiger Grammatik zerlegt `Tokenizer` eine Datei in Tokens und reicht diese an `ScriptParser` weiter. Der wiederum überprüft die Syntax der Datei und interpretiert gleichzeitig die Semantik der Tokensequenz. Bei einem Fehler wird eine Ausnahme „geworfen".

## Anhang: readme file

The CAV (Congestion Avoidance Visualization) Scripting Language:

### 1. Introduction

*"Congestion avoidance mechanisms allow a network to operate in the optimal region of low delay and high throughput, thereby preventing the network from becoming congested."*

\- [1]

The CAV – Scripting – Language is a scripting language for the CAV Tool. This tool visualizes vector diagrams like the ones explained in [1], but with the additional possibility of analyzing the asynchronous case and several additional features. These vector diagrams model the simple case of two senders sharing a single resource in a fair way; here, "fair" means equal sharing of this resource. It is used in order to implement your own congestion avoidance mechanisms (CAM) and afterwards test them in the CAV – Tool. Since congestion avoidance (CA) can become very difficult to describe mathematically in the asynchronous case, the CAV – Tool can be used to test a mechanism (synchronous or asynchronous) in a simple way.

I think the CAV – Scripting – Language is an easily understandable one and you learn a lot about CA if you know how the different mechanisms are implemented (although it's a simplified view of the problem).

The following sections describe how to use the CAV – Scripting – Language for the CAV – Tool . (both where developed by me for my baccalaureate – project; an earlier version of the CAV – Tool was developed by Michael Welzl who now supervises my project)

### 2. Description

#### 2.1 A Simple Example

Here is shown a script that describes the AIMD (additive increase multiplicative decrease) mechanism like it's used e.g. in TCP (linenumbers are not part of a real script!):

```
01:    if(traffic > 1){
02:          rate = rate * 0.5;
03:    }
04:    else{
05:          rate = rate + 0.1;
06:    }
```

In every script *traffic* and *rate* are predefined identifiers... they do not have to be initialized.

These identifiers are always initialized before running the interpreter of the CAV – Tool. These two are the only identifiers that are allowed two be used (and should be used) without initialization in a script – file. If you initialized these identifiers you would overwrite the values that where assigned in the CAV – Tool and your script would always come to the same result, ignoring all parameters you set in the CAV – Tool .

*if* is a keyword and a statement after *if* (always between “{“ and “}”) is only computed if the condition has a value greater than zero.

*else* is a keyword and only used together with *if* (*else* without *if* before is a syntax – error).

A statement after *else* (always between “{“ and “}”) is only computed if the condition of *if* was smaller than or equal to zero.

*traffic* is the sum of the rates of all senders (users) and *rate* is the rate of a single user.

If *traffic* equals 1, the available capacity is used. If *traffic* is greater than 1, the network is not efficient, but if *traffic* is less than 1 the network is congested and so not efficient. (The question arises why in the upper script the case that *traffic* equals 1 is not separated from the other cases. The answer is that in networks that are using AIMD, the feedback is binary; meaning, that a sender only gets the feedback that he should increase or decrease the rate. There are ideas of mechanisms that use exact feedback instead of binary, like CADPC, described in [2].)

In the script in line two you see that *rate* is reduced to the half if *traffic* is too high. If *traffic* could be higher, *rate* is increased by 0.1 (line 5). So *rate* is always increased in small steps until it is too high and then reduced … and it starts all over again.

## 2.2 Preprocessor – Directives

It is uncomfortable to change values (like 0.5 or 0.1 in the upper script) manually in the script itself if you want to test your script with different values. Therefore it's possible to predefine values with preprocessor – directives and in this way give them domains and optionally a default value.

The syntax for a preprocessor – directive is:

```
#define_param <name> range <min> to <max> [default <val>]
```

It means that you declare an identifier `<name>` with a minimum value of `<min>` and a maximum value of `<max>`. Optionally you can set the default value to `<value>` (otherwise the default value equals `<min>`).

Now the example from above with preprocessor – directives:

```
01:   #define_param alpha range from 0.1 to 0.9 default 0.2
02:   #define_param beta  range from 0.1 to 0.9 default 0.5
03:
04:   if(traffic > 1){
05:        rate = rate * beta;
06:   }
07:   else{
08:        rate = rate + alpha;
09:   }
```

If you test the script in the CAV – Tool it will now be possible to change the value of *alpha* and *beta*. Now you can test which values are ideal for *alpha* and *beta*.

## 2.3 Function Calls

There are several function calls supported in the CAV – Scripting – Language. They are all case – insensitive (you could type `max(a,b)`, `Max(a,b)` or `MAX(a,b)` or any other combination of upper and lower case letters to call the max – function that returns the maximum of two values) and recognized by the parser because of parentheses ("(" and ")") directly after the function name (`maX(a,b)` is a correct function call but `max (a,b)` is a syntax error because of the space between `max` and "(").

Supported functions are the following:

`abs(a)`, `cos(a)`, `exp(a)` {euler = `exp(1)`}, `int(a)` {nearest integer to a}, `max(a,b)`, `min(a,b)`, `pow(a,b)`, `random()` {a random number from the interval [0.0, +1.0) }, `sin(a)`, `sqrt(a)` and `tan(a)`

Now the example from above with function calls:

```
01:   #define_param alpha range from 0.1 to 0.9 default 0.2
02:   #define param beta  range from 0.1 to 0.9 default 0.5
03:
04:   maxrate = 1.0; # rate can never be higher than 100%
05:
06:   if(traffic > 1){
07:        rate = rate * beta;
08:   }
09:   else{
10:        rate = rate + alpha;
11:        rate = min(rate, maxrate);
12:   }
```

In this example the rate can never be higher than 1.0 (like in reality). For other scripts it could be necessary to define a *minrate* because the rate can never be less then 0.0 but in this example *rate* could never be negative in order to the domain of *beta*.

## 2.4 Comments

As you have already seen above, you can use “#” to start a single line comment. It's not possible to write a comment in the same line as a preprocessor – directive. Also, note that if there are empty lines or comment – lines, after a line with an error, the line number of the error – message in the CAV – Tool is the number of the line with the last empty line (every line without a statement in it is an empty line e.g. a line that only includes “}” is an empty line) or comment – line. (Since the parser throws the error, when it finds the next token and empty lines or comments are no tokens.)

## Literatur

[1] D. Chiu and R. Jain, Analysis of the Increase/Decrease Algorithms for Congestion Avoidance in Computer Networks, *Journal of Computer Networks and ISDN*, 17(1), 1989, 1-14.

[2] M. Welzl, Vector Representations for the Analysis and Design of Distributed Controls

[3] Guido Krüger, Handbuch der Java – Programmierung, *3. Auflage Addison – Wesley*, 2002.

[4] Bjarne Stroustrupp, Die C++ Programmiersprache, *4. überarbeitete Auflage Addison – Wesley*, 2000.