Common Logging Interface – Ein System zum Sammeln und Verarbeiten von Debugnachrichten in verteilten Umgebungen

Raimar Falke, Michael Peter, Achim Gratz und Rainer G. Spallek

Technische Universität Dresden
Institut für Technische Informatik
Mommsenstraße 13
D-01062 Dresden
{falke,peter,gratz,rgs}@ite.inf.tu-dresden.de
http://www.inf.tu-dresden.de/TU/Informatik/TeI/

Zusammenfassung. Mit CLI wird ein in Java implementiertes System vorgestellt, welches mit Hilfe eines minimalen API das zeitnahe Beobachten von mehrfädigen und verteilten Anwendungen auf der Basis von Debugnachrichten gestattet. Die Anwendung muß dazu nicht unter einem in-process Debugger laufen. Anwendung und Beobachter sind soweit als möglich entkoppelt. Die Selektion der Debugnachrichten wird mittels mehrstufiger Filterung anhand verschiedener Kriterien zur Laufzeit vorgenommen. Die Weiterleitung der Nachrichten erfolgt durch eine Transport-Schicht, die je nach vorliegender Umgebung angepaßt und ausgetauscht werden kann.

1 Einführung

Debugging in verteilten Systemen ist mit den herkömmlichen Mitteln nur schwer zu realisieren. Insbesondere ist die exakte Reproduktion von Programmzuständen in einem weiteren Programmlauf praktisch unmöglich. Das gilt teilweise auch für MT-Programme (*multithreaded*). Es existieren unseres Wissens zur Zeit keine frei verfügbaren Tools für Java-Applikationen, die sowohl verteilte als auch MT-Programme debuggen können. Eine kommerzielle Lösung ist mit *JWatch*[1] erhältlich. Einige freie und kommerzielle Entwicklungsumgebungen bieten immerhin partielle Unterstützung für MT-Programme.

Für ein am Institut für Technische Informatik in Entwicklung befindliches verteiltes Informations- und Aktionssystem wurde deshalb nach einer Debugschnittstelle mit folgenden Eigenschaften gesucht:

1. Zeitnahe und möglichst zeitrichtige Beobachtung von Anwendungen mit Hilfe von Debugnachrichten.
2. Im Falle der Nichtbenutzung ohne Neuübersetzung der Anwendung möglichst vollständig abschaltbar.

[1] http://www.JWatch.com

3. Einfaches API zur Einbindung in die Anwendung.
4. Möglichst wenig Rückwirkungen zwischen Debugger und Anwendung, insbesondere wenn eine der beiden Komponenten fehlerhaft ist.
5. Automatische Generierung von Metadaten wie Zeitstempel und Information über den Aufrufer (z.B. Klassen- und Instanz-Informationen, Stacktrace).
6. Filterung der Debugnachrichten sowohl an der Quelle (insbesondere Unterdrückung bestimmter Nachrichten) als auch an der Senke nach dynamisch vorgebbaren Kriterien.

Die Verwendung dieser Debugschnittstelle ist wie folgt vorgesehen: die im Netz verteilten Anwendungen senden ihre vorgefilterten Nachrichten an einen oder mehrere Debugger. Der Einsatz mehrerer Debugger ist beispielsweise dann sinnvoll, wenn hohe Verfügbarkeit zu gewährleisten ist, Lastverteilung erforderlich ist oder mehrere Personen dieselbe Anwendung debuggen möchten. Der Debugger nimmt die endgültige Filterung und Visualisierung der Daten vor und steuert auch die Filter in den jeweiligen Anwendungen.

Aus diesem Anwendungsszenario resultieren weitere Nebenforderungen wie z.B. Unabhängigkeit von der verwendeten Transportmethode. Eine Komprimierung oder Verschlüsselung der Debugnachrichten kann in Netzen mit niedriger Übertragungsbandbreite bzw. dem Debuggen von Anwendungen mit sensitiven Daten erforderlich werden. Für die *post-mortem-* und *offline-*Analyse ist die Protokollierung von Debugnachrichten in eine Datei wünschenswert. Diese Protokolldaten können beispielsweise auch der Generierung automatischer Tests dienen.

2 Implementierung

Für die unmittelbaren Ansprüche würde das aus obigen Forderungen resultierende System zu komplex werden. Daher wurden im Rahmen einer prototypischen Implementierung zunächst nur die wesentlichen Bestandteile realisiert, die fehlende Funktionalität kann bei Bedarf nachträglich hinzugefügt werden und erfordert keine Neuübersetzung der Anwendung. Beispielsweise wurde auf die Komprimierung des Datenstroms verzichtet, da ein schnelles Netz zur Verfügung steht, eine Verschlüsselung der Daten ist ebenfalls nicht erforderlich. Des weiteren wurde die Implementierung eines Werkzeuges für die dynamische Konfiguration zurückgestellt. Die Filterkonfiguration wird zur Zeit statisch über Konfigurationsdateien vorgenommen.

Das Gesamtsystem wird durch ein Schichtenmodell in relativ unabhängige Teilsysteme gegliedert. Dadurch ist ein Austausch der Implementierung gewisser Teilsysteme unter Beibehaltung der Schnittstellen möglich, wodurch Erweiterungen und Anpassungen des Funktionsumfanges unterstützt werden. Es wird zwischen der Anwendungsschicht, einer Highlevel-Schicht, einer Midlevel-Schicht, sowie der Transportschicht unterschieden. Die in Abbildung 1 dargestellte Architektur kennzeichnet den Iststand der Implementierung, d.h. es existiert noch keine Anwendungskomponente auf der Debugger-Seite.

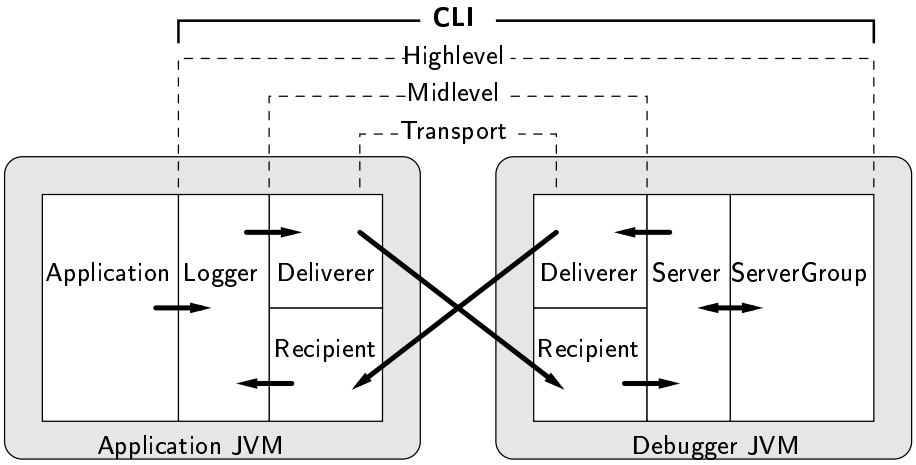

Abbildung1. Architekturmodell der Debugschnittstelle

In der Highlevel-Schicht finden die Steuerung und Filterung statt, außerdem werden bei der Einspeisung einer Nachricht die Metadaten hinzugefügt. In der Midlevel-Schicht werden die Nachrichten in getrennten Sende- und Empfangswarteschlangen gesammelt und die Kommunikation mit der Transportschicht abgewickelt, die eine transparente, gesicherte Verbindung bereitstellen muß. Dies ist bei Verwendung verbindungsloser oder ungesicherter Netzwerkprotokolle wie UDP[2] zu beachten. In allen Schichten von CLI werden *threads* verwendet, um eine zeitnahe Bearbeitung asynchroner und nebenläufiger Ereignisse zu gewährleisten. Dies erhöht auch die Skalierbarkeit auf Mehrprozessor-Rechnern.

Im Modell (siehe Abbildung 1) von CLI gibt es das Anwendungsprogramm (Client), welches mit Hilfe der Klasse tpv.cli.Logger mit dem CLI kommuniziert. Diese Klasse fügt die Metadaten hinzu, filtert entsprechend der Filterkonfiguration und verpackt die Nachricht in ein Transportobjekt, welches in eine Warteschlange eingereiht wird. Das wartende Transportobjekt wird von der Midlevel-Schicht adressiert und an die Transportschicht weitergegeben. Auf der Serverseite existiert eine gespiegelte Version der Transport- und Midlevel-Schicht. Für jeden Client wird ein tpv.cli.Server instanziiert, der die Daten an die tpv.cli.Servergroup übergibt, wo sie dann entsprechend der dortigen Filterkonfiguration weiterverarbeitet werden. Die Schnittstelle zwischen Anwendungsprogramm und CLI ist die Klasse tpv.cli.Logger. Die Verbindung zwischen Midlevel und Transportschicht wird durch die Schnittstellenbeschreibungen (interface) tpv.cli.transport.Transmitter für den Sender und tpv.cli.transport.Receiver für den Empfänger vorgegeben. Die Klassen des Paketes tpv.cli.filter werden schrittweise aus verschiedenen Interfaces und deren Implementierungen aufgebaut.

[2] Siehe RFC 768 und Nachfolger

3 Probleme und Erfahrungen

Zunächst wurde versucht, für jede Nachricht eine neue TCP-Verbindung[3] aufzubauen. Da die *sockets* nach dem Schließen nicht kontinuierlich freigegeben wurden, wurden die Filedeskriptoren im JVM-Prozeß (JVM – *Java Virtual Machine*) schnell verbraucht. Dadurch kam kein stabiler Betrieb zustande und es gingen Nachrichten verloren. Verschiedene Implementierungen der JVM zeigten hier im Detail unterschiedliches Verhalten. Dieses Problem wurde gelöst, indem die einmal aufgebaute Verbindung für mehrere Nachrichten benutzt wird. Dadurch kann außerdem der Overhead des Verbindungsauf- und -abbaus über viele Nachrichten amortisiert werden.

Die Midlevel-Schicht geht von einem fehler- und verlustfreien Transport der Transportobjekte aus. Es existiert je eine Beispielimplementierung der Transportschicht für die Internet-Protokolle TCP und UDP, es ist aber auch vorstellbar, die Informationen per NFS[4] oder RMI[5] zu übertragen, oder zur Steigerung der Leistung eine Transportschicht native zu implementieren. UDP ist im Gegensatz zu TCP ein verbindungsloses Protokoll, bei dem der Empfang der Daten nicht gesichert ist. Daher muß die Sicherung von einem Teil des CLI übernommen werden, während diese bei TCP effizient vom Betriebssystem übernommen wird. Der implementierte UDP-Transport ist daher langsamer als der TCP-Transport.

Sowohl die Midlevel-Schicht als auch die Transportschicht wandeln ihr jeweiliges Transportobjekt durch Serialisierung[6] in einen Bytestrom um. Aus diesem Grund ist abzusehen, daß ein mehrschichtiger Aufbau eine Vergrößerung des Datenvolumens sowie der Ausführungszeit nach sich zieht. In der vorliegenden Implementierung beträgt der Overhead durch mehrfache Serialisierung pro Nachricht insgesamt etwa 200 Byte bei einer typischen Netzwerkpaketgröße von circa 700 Byte für eine Nachricht.

Wenn ein Logger ein Fehlverhalten zeigt, kann der Verbindungsabbruch vom Server unter Umständen nicht erkannt werden und zu Fehlfunktionen sowohl beim Server als auch bei den nicht betroffenen Clienten führen. Die Behebung solcher Fehler gestaltet sich schwierig, da verschiedene Fehlerursachen gleiche Auswirkungen haben können (z.B. kann nicht zwischen einer unterbrochenen Netzwerkverbindung und dem Absturz der Applikation unterschieden werden). Eine Reihe von Fehlern könnten außerdem nur durch Exceptions an die Anwendung gemeldet werden, die diese dann behandeln müßte. Dies wurde in dem vorliegenden API aber nicht vorgesehen, um dieses minimal zu halten. Wenn das Problem erkennbar temporärer Natur ist (durch Tests im CLI ermittelbar), sollte im CLI eine für die Anwendung transparente Fehlerbehandlung erfolgen.

CLI selber kann sehr gut partiell debuggt werden, es erzeugt Debug-Nachrichten auf der Konsole. Die Ausgaben von einzelnen Paketen oder Klassen lassen sich in verschiedenen Stufen ohne Neuübersetzung durch Aufrufparameter zu- und abschalten.

[3] Siehe RFC 793 und Nachfolger
[4] Siehe RFC 1094 und Nachfolger
[5] http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/
[6] http://www.javasoft.com/products/jdk/1.1/docs/guide/serialization/

4 Ergebnisse und Ausblick

4.1 Performance

Es wurden eine Reihe von Testläufen in verschiedenen Konfigurationen durchgeführt. Die in Tabelle 1 angegebenen Zeiten (in Millisekunden) sind der Durchschnitt von zehn Testläufen (der beste und schlechteste Durchlauf wurden gestrichen) mit jeweils 1000 Aufrufen der Methode tpv.cli.Logger.log(). Die Rechner A, B und C laufen unter Solaris 2.6 (A, B: Ultra 2/2300 mit 256 MB RAM; C: Ultra Enterprise 450/3300 mit 512 MB RAM), der Computer D unter Windows 95 (PentiumII/233 mit 64 MB RAM). Die Netzwerk-Verbindung wurde über ein *switched* GigabitEthernet (A, B, C) bzw. *switched* FastEthernet (D) realisiert.

Da beim Verbindungsaufbau mit dem Server ein *handshake* durchgeführt wird, dauert ein solcher max. eine Sekunde. Das Schließen der Verbindung kann innerhalb einer drittel Sekunde geschehen, sofern nicht noch Nachrichten in der Warteschlange stehen. Das Hinzufügen der Metadaten und Verpacken der Nachricht in ein Transportobjekt verbrauchen im besten Fall weniger als 2 ms. Die meiste Zeit wird für den Transport der Nachricht benötigt, dabei ist jedoch nicht das Netzwerk der begrenzende Faktor. Sowohl die erzielte Bandbreite von etwa 570 kBit/s als auch die Paketrate von circa 100 Pkts/s liegen weit unter den Möglichkeiten des Netzwerkes. Eine Untersuchung, wo genau die Differenz zu den maximal möglichen 3,5 MBit (bei einer Paketgröße von 700 Byte) und 625 Pkts/s (Test „leerer Filter“) verursacht wird, steht noch aus, erste Experimente haben jedoch ergeben, daß vermutlich kein einzelner Engpaß für den Leistungsverlust verantwortlich ist. Die Unterschiede durch Verwendung der verschiedenen JVM-Versionen und zwischen den einzelnen Konfigurationen deuten darauf hin, daß sowohl Ineffizienzen bei der Serialisierung als auch bei der Weiterleitung der Nachrichten in und durch die Transportschicht in Betracht kommen.

Die Werte in Tabelle 1 sind wie folgt zu interpretieren:

Create ist die Zeit zum Erzeugen eines tpv.cli.Logger, dies schließt bei einer Filterkonfiguration mit Weiterleitung von Nachrichten auch einen Verbindungsaufbau zum Server mit ein. In allen anderen Fällen wird kein Server kontaktiert.

Log ist die Zeit für einen Aufruf der Funktion tpv.cli.Logger.log() aus der Sicht der Anwendung.

Close ist die Zeit, in der CLI alle noch wartenden Nachrichten verarbeitet sowie die evtl. bestehende Verbindung zum Server abbaut.

Folgende Konfigurationen wurden getestet:

abgeschaltet: Das gesamte CLI wurde mit Hilfe von -Dtpv.cli.disable=true für diese JVM deaktiviert.

leerer Filter: Es werden die Metadaten ermittelt und die Nachricht in ein neues Transportobjekt verpackt, welches dann vom Filter verworfen wird.

Tabelle 1. Ergebnisse der Testläufe

Konfiguration	JVM	Quelle-Ziel	Create	Log	Close	Gesamt
abgeschaltet	1.1.6 JIT	A	16	0,1	1	70
abgeschaltet	1.1.6	A	11	0,1	0	76
abgeschaltet	1.1.5	D	10	0,1	0	490
leerer Filter	1.1.6 JIT	A	100	1,6	1	1747
leerer Filter	1.1.6	A	67	2,5	1	2683
leerer Filter	1.1.5	D	110	5,5	0	5990
10 Strings	1.1.6 JIT	A	195	2,8	0	3107
10 Strings	1.1.6	A	139	4,4	0	4638
10 Strings	1.1.5	D	220	8,0	0	8460
kurz speichern	1.1.6 JIT	A	177	3,7	1	3964
kurz speichern	1.1.6	A	122	5,9	0	6111
kurz speichern	1.1.5	D	160	12,1	0	12790
lang speichern	1.1.6 JIT	A	174	5,7	1	5949
lang speichern	1.1.6	A	118	11,0	0	11212
lang speichern	1.1.5	D	160	29,7	0	30150
weiterleiten	1.1.6 JIT	A–A	842	13,1	5777	19881
weiterleiten	1.1.6	A–A	577	23,3	4381	28293
weiterleiten	1.1.5	D–D	920	79,1	15000	96620
weiterleiten	1.1.6 JIT	A–B	881	9,9	4073	14967
weiterleiten	1.1.6	A–B	693	16,8	2614	20177
weiterleiten	1.1.5 / 1.1.6 JIT	D–B	930	65,7	220	67500
2 Clienten	1.1.6 JIT	A,B–C	759	17,2	9768	27768
2 Clienten	1.1.6	A,B–C	609	21,2	13594	35324

10 Strings: Es werden die Metadaten ermittelt und die Nachricht in ein neues Transportobjekt verpackt, welches dann vom Filter gegen 10 nicht passende Strings verglichen und somit verworfen wird.

kurz speichern: Die Filterdatei besteht nur aus einer saveShort Anweisung, d.h. die Nachricht wird in eine einzeilige lesbare Form konvertiert und in eine lokale Datei gespeichert, aber nicht an die Transportschicht übergeben.

lang speichern: Die Filterdatei besteht nur aus einer saveLong Anweisung, d.h. die Nachricht wird in eine mehrzeilige lesbare Form konvertiert und in eine lokale Datei gespeichert, aber nicht an die Transportschicht übergeben.

weiterleiten: In dieser Konfiguration besteht die Filterdatei aus einem einzigen forward-Befehl, alle Nachrichten werden an die Transportschicht übergeben und an den Server weitergeleitet. Die Tests wurden mit Server und Client auf demselben Rechner als auch mit dem Server auf einem entfernten Rechner durchgeführt.

2 Clienten: Diese Konfiguration ist identisch zur vorangegangenen, jedoch sind gleichzeitig zwei Clienten mit dem Server verbunden, der auf einem entfernten Rechner läuft.

Die Ergebnisse zeigen, daß CLI im abgeschalteten Zustand praktisch keine Ressourcen verbraucht und durch *just-in-time compilation* (JIT) teilweise signifikante Leistungsgewinne erzielt werden. Die Filterung der Nachrichten auf der Anwendungsseite verbraucht weniger Zeit als das Speichern oder Senden der Nachricht. Bei der Interpretation der Ergebnisse ist zu beachten, daß der Test-Client kontinuierlich mit der höchstmöglichen Rate Nachrichten generiert. Die Last unter realen Bedingungen ist auch bei mehreren Clienten nicht kontinuierlich. Die Warteschlangen haben sich als effektives Mittel zum Abfangen solcher Lastspitzen erwiesen und sorgen im Normalbetrieb für eine zügige Rückkehr zum Anwendungsprogramm. Bei unseren Tests traten dagegen teilweise erhebliche Rückstaueffekte auf. Die in Tabelle 1 aufgeführten Werte sind deshalb pessimistisch, jedoch traten im (seltenen) Einzelfall auch erheblich größere Verzögerungen auf, die evtl. durch Kontextwechsel verursacht werden. Wir erwarten, daß ein einzelner, entsprechend leistungsfähiger Server mit mehr als zehn Clienten belastet werden kann, wobei jedoch auch die Komplexität der Filter eine Rolle spielen wird.

4.2 Vergleich mit JMS

Gegen Ende der Implementierungsarbeit wurde auf der JavaOne-Konferenz das Java™ Message Service API (JMS[7], basierend auf JNDI[8]) vorgestellt. In Tabelle 2 sind wichtige Merkmale von CLI und JMS gegenübergestellt.

Tabelle 2. Vergleich von CLI und JMS anhand wichtiger Merkmale

Kriterium	CLI	JMS
Unterstützte Verbindungen	Punkt-zu-Punkt	Punkt-zu-Punkt, *Publisher-Subscriber*
Zielangabe	IP-Adresse	*administered objects* im Namensraum des JNDI
Message-Header	Header mit festem Format	Header mit benutzerdefinierten Properties
Message-Body	String mit Debug-Message	fünf Nachrichtenformate
Filterkriterien	nach Metadaten und Message-Body	nach Metadaten und Properties
Filterrealisierung	proprietäre Lösung mit Filtersets	SQL

Obwohl die Ausgangspunkte für die Entwicklung von JMS und CLI sehr verschieden sind, kann CLI recht einfach zu einer Beispielimplementierung von JMS

[7] http://www.javasoft.com/products/jms/
[8] http://www.javasoft.com/products/jndi/

erweitert werden. Umgekehrt wäre es ebenfalls möglich, CLI aufbauend auf JMS zu implementieren. Diese Überlegungen sind aber bis zur Verfügbarkeit von Java 1.2 zurückgestellt, da vorher keine gesicherten Aussagen zur Leistung von JMS zu machen sind.

4.3 Andere Transportmethoden

Durch den modularen Aufbau ist eine Auswechselung der Transportschicht einfach möglich. Mit TCP und UDP deckt CLI sowohl den verbindungsorientierten als auch den verbindungslosen Transport ab. Es ist zu vermuten, daß beispielsweise bei Austausch der TCP- gegen eine RMI-Schicht letztere langsamer sein wird, da zusätzlicher Overhead entsteht. Eine andere Möglichkeit besteht in der Reimplementierung der Midlevel-Schicht mittels RMI unter Wegfall der Transportschicht. Damit wären dann nur noch die RMI-Transportmethoden nutzbar, die aber bei entsprechend optimierter RMI-Implementierung u.U. einen Leistungsgewinn bedeuten würden.

5 Zusammenfassung

Mit CLI wird ein System vorgestellt, mit dem mehrfädige und verteilte Anwendungen auf der Basis von Debugnachrichten zeitnah und relativ zeitrichtig beobachtet werden können. Dabei hat sich gezeigt, daß durch den Schichtenaufbau das Gesamtsystem flexibel an neue Anforderungen angepaßt werden kann. Diese Flexibilität muß aber durch eine etwas geringere Leistung erkauft werden, was für interaktive Anwendungen tragbar erscheint. CLI stellt eine minimale Schnittstelle zur Anwendung bereit, die aus nur einer einzigen Klasse mit vier öffentlichen Methoden besteht. Es gibt keine Rückmeldungen von CLI, auf welche die Anwendung reagieren müßte. Dadurch ist es einfach möglich, bestehenden Debug-Code mit Ausgaben auf die Java-Konsole auf CLI umzustellen. Die Einbindung von CLI in bestehende und neue Anwendungen wird durch das nicht-intrusive API ebenfalls erleichtert.

Im Ergebnis der Testläufe ist zum Ausdruck gekommen, daß CLI in nicht zeitkritischen Anwendungen, wie sie interaktive Systeme in der Regel darstellen, erfolgreich benutzt werden kann. Die Robustheit des Systems gegenüber Fehlerzuständen besonders im Netzwerk ist noch zu steigern. Der durch CLI verursachte zusätzliche zeitliche Aufwand kann durch globales Abschalten von CLI über einen Aufrufparameter bzw. Setzen einer *property* zur Laufzeit praktisch auf Null reduziert werden. Dadurch ist es aus Sicht der Anwendungsleistung unnötig, den Debug-Code aus der fertigen Anwendung zu entfernen. In Einzelfällen werden jedoch der durch die Strings verbrauchte Speicherplatz und die Größe der .class- bzw. .jar-Dateien eine Rolle spielen. Die Entscheidung für eine mehrstufige Filterung der Nachrichten hat sich als richtig erwiesen, wobei die implementierte zweistufige Filterung leicht auf weitere Stufen erweiterbar ist.

A Beispiel einer Konfigurationsdatei für einen Filter

```
#Grammar:
# compareActions are =,!=,>,<,>=,<=,|=,|!
#    |= embodies a Regexp match
#    |! embodies a Rexexp mismatch
# Actions are:
#  - "forward"      forward to a server
#  - "displayShort" display one line local
#  - "displayLong"  display many lines local
#  - "saveShort"    save one line into a file
#  - "saveLong"     save many lines into a file
# Values are:
#  - Host[Name]
#  - VM[Name]
#  - ThreadGroup[Name]
#  - Thread[Name]
#  - Class[Name]
#  - Instance[Name]
#  - Msg
#  - Level[Name]
# simpleValue is a string or a number in "" or in ''
# a triple consist of a Value, acompareAction
#   and a simpleValue
# one line
#  is a comment if it starts with '#'
#  or a line containing only whitespaces, which is ignored
#  or (triple)* [+-] Actions (actionArgument)?
#  can be continued by having \
#  as the last element in line
#

#Samples:
# forward all messages except those whose loglevel is
# LEVEL_DEBUG to the server
LevelName != 'Debug' +forward

# show all messages that came from the second VM on localhost
# and a thread which is named IdleThread on the local console
HostName=localhost VM=2 ThreadName='IdleThread' \
    +displayShort

# save all messages from host "foobar"
# and from package "tpv.abc" to local file localLog
HostName="foobar" ClassName|="^tpv.abc" \
    +saveLong "localLog"
```

B Demonstrationsprogramm Demo.java

```
package tpv.cli;

import tpv.cli.Logger;               // Only this class is needed for inclusion of CLI

/**
 * Simple Demo for CLI.
 */
public class Demo
{
  Demo()
  {
    /**
     * There are four ways to construct a new Logger.  You must supply a reference to the
     * object, using this Logger and you can supply an alternative default loglevel and
     * the name of a file containing the filter configuration.  When no default loglevel
     * is specified, LEVEL_DEBUG is implied. */
    Logger log=Logger.getLogger(this);

    /**
     * Logger provides the method log() in various versions.  Simply log(String) logs this
     * String with the default loglevel.  If there were too many messages or the network
     * is too slow then log() may block, otherwise the call returns immediately and the
     * message is delivered in the background. */
    String str="SimpleMessage";
    log.log(str);

    /**
     * You also can specify a loglevel: log(byte,String), the levels are defined in
     * tpv.cli.Logger. */
    log.log(Logger.LEVEL_ERR, "There was an error.");

    /**
     * You can also log Throwable's with and without specifying a loglevel:
     * log(byte,Throwable) and log(Throwable) */
    try
    {
      Object temp=null;
      Class classOfTemp=temp.getClass();
    }
    catch(NullPointerException e)
    {
      log.log(Logger.LEVEL_CRIT, "There was an exception:");
      log.log(Logger.LEVEL_CRIT, e);
    }

    /**
     * Before you destroy the VM you must call close() to deliver all buffered messages.
     * You can also use the function flush(), but this function doesn't make much sense
     * when you also feed messages from another Thread to the Logger.*/
    log.close();

    /**
     * Now we can exit without any trouble.*/
    System.exit(0);
  }

  public static void main(String args[])
  {
    /**
     * We need a new instance of the class because the Logger needs an instance. */
    new Demo();
  }
}
```