# 1 Что такое система мониторинга?

Система мониторинга — это коллекторная часть и обработчик, в ее задачу входит не столько сбор информации, сколько ее обработка, принятие решений об оповещении или совершении каких-нибудь действий. При этом непосредственно коллекторная часть может быть внешней.

Обработка собранной информации может происходить одним из следующих способов:

- агрегация;
- создание новой информации по имеющейся;
- оповещение или совершение каких-либо действий.

# 2 Коллекторная часть

Задача коллектора в системах мониторинга — не столько сбор данных, сколько приведение их к какому-то внутреннему виду. Учитывая сколько чисто коллекторных систем сейчас существует (та же Ganglia) писать свою смысла особого нет; скорее надо внимательно осмотреть имеющиеся и выбрать наиболее подходящую.

## 2.1 `collectd`

Особый интерес представляет использование `collectd`. Это демон сбора информации о системе. Он не претендует на какую-то сложную функциональность (выдержка с сайта `http://collectd.org`):

> It does not do monitoring. The data is collected and stored, but not interpreted and acted upon.

Однако отличается несколькими важными качествами:

- распределенный сбор информации — несколько серверов могут отсылать друг–другу данные;
- достаточное количество уже написаных плагинов для сбора разного типа данных, от трафика до изменения частоты процессора;
- простой протокол обмена;
- простая база данных [1].

[1] Вообще говоря `rrd` не является базой данных в общепринятом понимании, однако для задач мониторинга базы с агрегацией устаревших данных вполне подходят.

Систему мониторинга можно подключить к `collectd` в качестве потребителя информации (при этом коллектор еще и обеспечит построение красивых графиков):

- через UNIX–сокет посылая запросы на нужную информацию; подробнее этот протокол описан в `collectd-unixsock(5)`
- получать multicast или unicast сообщения от других демонов `collectd`. [2]

Первый вариант лучше тем, что не надо парсить сетевые сообщения, второй — что изменения конфигурации автоматически отслеживаются.

## 2.2 Почему не `syslog`?

Вообще говоря, `syslog` представляет из себя весьма хороший «коллектор». По крайней мере, он обладает типичными признаками оного — сбор информации из разных источников (возможно, удаленных) и складирование ее в каком-то общем формате в каком-то одном месте. Тем не менее плодятся сотни систем, в изрядной степени дублирующие `syslog`. Почему?

Одна из причин — сложность «разбора» записей. В логах они лежат в весьма свободном текстовом формате. Без предварительного вчитывания в них можно только сказать, кто и когда создал эту запись (ну и уровень). Содержание же покрыто завесой мрака. К этому добавляется еще и то, что многие демоны либо не пользуются `syslog`, либо сваливают свои данные в отдельные файлы (например есть «устоявшийся» формат `xferlog` для FTP серверов).

# 3 Требования на Систему

Что хочется от системы мониторинга? Прежде всего она должна быть такой, чтоб ее можно было использовать, причем в распределенных структурах. Из этого уже вытекают два требования — распределенность и компактность. Компактной здесь называется система которая не требует установки полусотни дополнительных пакетов, запуска 3-5 демонов и кучи места на диске. Кроме этого, система должна поддерживать свои базы данных собственными силами (и не иметь зависимостей на СУБД, типа MySQL, pSQL и тд), например `rrd`, `sqlite`, `bdb`.

## 3.1 Требования

- распределенность, возможность обменитьваться данными с другими, отсутствие центральной незаменимой управляющей точки
- компактность, возможность хранить данные средствами системы

[2] В данный момент реализован клиент к `collectd`–серверу, обрабатывающий поток результатов и приводящий к внутреннему виду. Подробнее описано в 5

- расширяемость, возможность добавлять новые типы/источники в работающую систему; расширяемость может быть достигнута за счет того, что система не имеет привязки к какому-то конкретному типу данных, а работает с абстрактными сообщениями вида «работает», «не работает», «недоступно», ...
- внешний интерфейс или удобоваримый формат хранения данных

## 3.2 Откуда берется информация?

- активные запросы
- пассивный сбор информации от какой-то другой системы сбора
- прием оповещений (типа SNMP Trap)
- обмен информацией с другими

Например, возможна минимальная установка системы без коллекторной части и без конфигурирования, если она будет забирать информацию из кластерной системы очередей (`PBS`, `SGE`, ...), из кластерной системы мониторинга (`Ganglia`) или из какой-нибудь коллекторной системы типа `collectd`.

### 3.2.1 Zero Configuration

Система может вести себя похожим на `collectd` образом, а именно, принимать все неизвестные данные и обрабатывать их. Зависимости, видимо, при этом установить не удастся.

## 3.3 Зависимости

В системе должен быть функционал обработки зависимых событий. Например, отключение какого-то узла может быть из-за обрыва линка где-то значительно выше. Поскольку зависимости пишутся в виде «А зависит от Б и (В или Г)» возможна обработка только «снизу–вверх», то есть проверка вверх по зависимостям, какой-же реально компонент привел к выходу из строя системы. Однако это возможно лишь в системах с активными запросами. В случае пассивных опросов (с какой-то периодичностью обновляется информация), возможно «опускание» данных, например, в примере выше, выход из строя Б приведет к отметке на А, что оно тоже не может функционировать.

## 3.4 Гипотезы и корреляции

В автоматических системах зачастую не хватает двух вещей — гипотез и корреляций. **Гипотезой** будем называть следующее. Пусть есть зависимость $C \to B \to A$

(где $B \to A$ обозначает, что $B$ зависит от $A$). В случае если из строя выходит $C$ оператор сразу предполагает, что возможно это произошло из-за сбоя $A$ или $B$. То есть, появляются гипотезы $A^?$ и $B^?$.

**Корреляцией** (или связанными сбоями) будем называть сбои, происходящие в одно и то же время по одной и той же причине, но связь между которыми нельзя установить располагая лишь информацией известной системе. Например, отключение питания в здании приведет к отключению всего и вся (на упсах сколько-то протянет, но тоже выключится), при этом никаких зависимостей между узлами скорее всего найти не удастся. Это задача не очень сложная для оператора, но довольно тяжелая для автоматических систем.

# 4 Гипотезы

Система обработки гипотез может выглядеть следующим образом. Пусть существует некое множество гипотез $\mathbb{A} = \{A_i^?\}$ (вообще говоря для каждого объекта $A_j$ в системе существует гипотеза $A_j^?$ о том, что он не работает). Система работает с множеством пар $(a, p) \in \mathbb{A} \times (0, 1]$, где $a$ — гипотеза, а $p$ — вероятность того что она верна. Каждое приходящиее в систему событие (сообщение о том, что что-то работает или не работает) может порождать или опровергать гипотезы. Например если для узла $A_i$ приходит сообщение о том, что он работает нормально (обозначим его $A_i^+$), гипотеза $A_i^?$ и все выше неё по дереву будет удалено, ибо она не верна. Если же приходит сообщение $A_i^-$, то нужно породить и проверить гипотезы о сбоях на узлах, от которых зависит $A_i$ и, если они не верны, объявить $A_i^?$ истинной. В случае если порождается какая-то гипотеза, но она уже находится на обработке, ее вероятность надо повысить. Таким образом в системе все время находится некоторый список гипотез с приписаными вероятностями. В случае, если вероятность превысила критическую, система оповещает оператора.

Вообще говоря у гипотезы существует еще и временная метка, когда она была создана. В случае, если слишком долго не удалось ее опровергнуть тоже необходимо оповестить оператора. Также, возможно повышение вероятности со временем (в разумных пределах).

## 4.1 Сообщения

Типы сообщений в системе бывают следующими:

- «узел $A_i$ работает» — узел $A_i$ не ломался, восстановился или стал доступен;
- «узел $A_i$ не работает» — сообщение о том что именно $A_i$ узел **гарантировано** не работает;
- «узел $A_i$ недоступен» — информация об узле по каким-то причинам не может быть получена. Сообщение не указывает напрямую на то, что не работает именно $A_i$. Ниже «сбоем» называется именно сообщение о недоступности.

Следует различать сообщения «недоступен» и «не работает», хотя для сетевого мониторинга зачастую они путаются. Например, ответ на ICMP запрос «Destination Host Unreachable» не является индикатором того, что хост не работает, хотя по адресу отправителя этого ICMP ответа можно предположить на каком участке сбой.

### 4.1.1 Преобразование информации

Когда данные из какого-то числового вида приводятся к абстрактном виду описанному выше? Это преобразование специфично для каждого узла и типа данных (если все делать правильно), или хотя бы только для типа данных. Например, пинг со временем отклика больше 10 секунд в большинстве случаев сигнализирует о том, что система находится на грани сбоя. Хотя для некоторых сетей этот показатель может считаться нормой. При этом, та же цифра «10» может быть вполне себе приемлимой загрузкой процессора.

Поскольку такое преобразование должно точно настраиваться под конкретную структуру, необходимо что бы система предоставляла возможность задавать правила внешним образом (конечно, это не исключает необходимости в стандартном наборе правил приведения). В этом плане подход Flame с заданием функций заслуживает внимания.

## 4.2 Вероятности

### 4.2.1 Обозначения

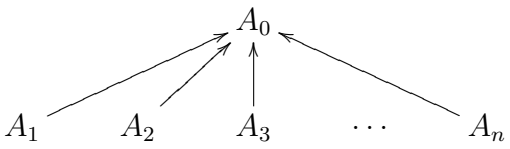
- $A_i$ — вершины дерева $T$.
- $p_i$ — вероятность (точечная) сбоя для узла $A_i$.
- $A_i^?$ — простая гипотеза о том, что $A_i$ сломался (именно сломался, а не недоступен).
- $A_{i,\ldots,j}^? = A_i^? \wedge \ldots \wedge A_j^?$ — сложная гипотеза об одновременном сбое $A_i, \ldots, A_j$. При этом должно соблюдаться условие, что ни одна $A_k$ не имеет зависимостей от $A_l$.
- $H(T)$ — множество гипотез в дереве $T$.
- $P(A^?)|_T$ — вероятность истинности гипотезы $A^?$ в дереве $T$.
- $\hat{P}(A^?)$ — нормированная вероятность. То есть если появилась информация о сбое, значит хотя бы одна гипотеза верна и сумма нормированных вероятностей равна 1.

### 4.2.2 Построение дерева

Рассмотрим две «образующих» структуры в дереве.

$$A_n \longrightarrow A_{n-1} \dashrightarrow A_1 \longrightarrow A_0 \tag{1}$$

(2)




Из этих двух структур можно построить произвольное дерево, при этом (1) введена для уменьшения количества шагов построения. Рассмотрим эти случаи и посчитаем для них распределения вероятностей гипотез.

**Случай (1).** Пусть произошло событие $A_n^-$. Оно породило гипотезы $A_i^?$ с вероятностями (ненормированными)

$$P(A_0^?) = p_0,\ P(A_1^?) = (1-p_0)\cdot p_1,\ \ldots,\ P(A_i^?) = \prod_0^{i-1}(1-p_j)\cdot p_i$$

Сумма этих вероятностей меньше 1

$$p_0 + p_1(1-p_0) + \ldots + p_i\prod_0^{i-1}(1-p_j) = 1 - \prod_0^{i}(1-p_j)$$

Поскольку мы знаем, что сбой есть (хотя бы один), получим нормированные вероятности

$$\hat{P}(A_i^?) = \frac{1 - p_i\prod_0^{i-1}(1-p_j)}{1-\prod_0^n(1-p_j)}$$

**Случай (2).** Пусть произошли сбои $A_1, \ldots, A_n$. Рассматривается одновременный сбой всех узлов, потому что нам интересно лишь та часть дерева, про которую у нас есть информация. Если про какую-то листовую вершину $A_k$ мы ничего не знаем, то и в дереве она не будет представлена.

$$P(A_0^?) = p_0,\ P(A_{1,\ldots,n}^?) = (1-p_0)\prod_1^n p_i$$

$$\hat{P}(A_0^?) = \frac{p_0}{1-(1-p_0)(1-\prod_1^n p_i)},\ \hat{P}(A_{1,\ldots,n}^?) = \frac{(1-p_0)\prod_1^n p_i}{1-(1-p_0)(1-\prod_1^n p_i)}$$

Как несложно заметить, в случае $n = 1$ случаи (1) и (2) эквивалентны.

**Композиция.** Пусть есть какое-то дерево $T$, покажем как его листовую вершину $A_k$ можно заменить на поддерево $T'$.

**Утверждение 4.1.** *Вероятность $P(A^?_{\dots,k,\dots})$ в дереве $T$ имеет вид $p_k \cdot f(T \setminus A_k)$, где в двухкомпонентном графе $T \setminus A_k$ рассматривается только компонента содержащая корень.*

*Доказательство.* Следует из независимости событий и соображений размерности ;) □

**Следствие 4.2.** *Вероятность $P(A^?_{i,\dots,j})$ имеет вид $\prod_{i,\dots,j} p_k \cdot f(T')$, где $T'$ — дерево $T$ без всех вершин $A_i, \dots, A_j$.*

Для этого необходимо для дерева $T'$ посчитать его «вес» $W(T')$, то есть сумму ненормированных вероятностей для гипотез в $T'$. Эта величина всегда меньше 1. Соответственно, это вероятность «сбоя» в поддереве $T'$. Положим $p_k = W(T')$ и пусть вероятность гипотезы $A^?_k$ (в дереве $T$) $P(A^?_k) = p_k \cdot f(T \setminus A_k) = W(T') \cdot f(T \setminus A_k)$, а вес дерева $W(T) = w(p_0, \dots, p_k, \dots) = w(p_0, \dots, W(T'), \dots)$.

Рассмотрим дерево $T'' = T \cup_{A_k} T'$, то есть дерево $T$ у которого листовая вершина $A_k$ заменена на дерево $T'$. Рассмотрим подмножество $H_{A_k}(T)$ множества гипотез $H(T)$ такое, в которое входят все гипотезы из $H(T)$ содержащие предположение о сбое узла $A_k$. Тогда множество $H(T'')$ это $H(T \setminus A_k) \cup (H_{A_k}(T) \times H(T'))$, то есть каждое вхождение предположения о сбое в $A_k$ заменяется множеством гипотез о сбое в поддереве $T'$.

Посчитаем вероятности в дереве $T''$. Для гипотез из $H(T \setminus A_k)$ они не изменятся (следует из 4.1). Рассмотрим теперь гипотезы из $H_{A_k}(T) \times H(T')$. Они имеет вид $A_{i_0,\dots,i_n,j_0,\dots,j_k} = A^?_{i_0,\dots,i_n} \wedge A^?_{j_0,\dots,j_k}$, где первая компонента (обозначим их для краткости $A^?_0$ и $A^?_1$) принадлежит дереву $T \subset A_k$, а вторая — $T'$. Поскольку гипотезы независимы, вероятность $P(A_{i_0,\dots,i_n,j_0,\dots,j_k}) = f(A^?_0) \cdot g(A^?_1)$.

**Утверждение 4.3.** *$g(A^?_1) = P(A^?_1)$ в дереве $T'$.*

*Доказательство.* Не доказательство, но рассуждение...

Просуммируем $P(A^?_0 \wedge A^?_1)$ по всевозможным $A^?_1$. Поскольку они независимы, а $P(A^?_0 \wedge A^?_1)$ представимо в виде $f(A^?_0) \cdot g(A^?_1)$, то сумма будет равна $f(A^?_0) \sum g(A^?_1)$. Поскольку результат это $P(A^?_0 \wedge A^?_k)$, а $g(A^?_k)$ равна $c \cdot p_k$. Константу можно вынести в $f(A^?_0)$ и получить, что $\sum g(A^?_1) = p_k = W(T')$. □

**Пример**. Пусть дерево $T$ состоит из двух вершин $A_0$ и $A_1$, дерево $T'$ — из $A_1$ и $A_2$. Тогда $p'_1 = W(T') = 1 - (1 - p_1)(1 - p_2)$, $W(T) = p_0 + (1 - p_0) \cdot (1 - (1 - p_1)(1 - p_2)) = 1 - \prod_0^2 (1 - p_i)$, вероятности гипотез

$$\begin{aligned}
P(A^?_0)|_T &= p_0 = P(A^?_0) \\
P(A^?_1)|_T &= (1 - p_0) \cdot p'_1 = (1 - p_0) W(T') = (1 - p_0)(P(A^?_1)|_{T'} + P(A^?_2)|_{T'}) \\
P(A^?_1) &= (1 - p_0) P(A^?_1)|_{T'} = (1 - p_0) \cdot p_1 \\
P(A^?_2) &= (1 - p_0) P(A^?_2)|_{T'} = (1 - p_0) \cdot p_2 (1 - p_1)
\end{aligned}$$

Как видим, это совпадает со случаем 1.

## 4.3 Расширение на графы

В реальности часто встречаются структуры, которые нельзя описать деревом, например когда узел $A_i$ зависит и от узла $A_j$, и от $A_k$. Для этого нам придется несколько модифицировать изложенные выше.

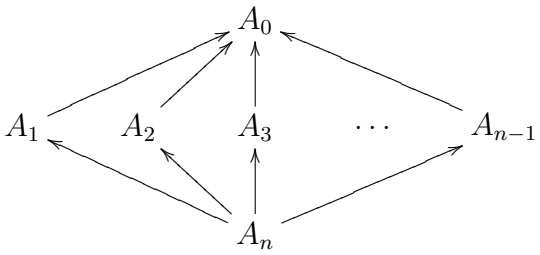
### 4.3.1 Обозначения

Некоторые новые обозначения

- $G$ — граф.
- $\hat{A}^?_{i,\ldots,j}$ — сложная гипотеза вида $A^?_i \vee \ldots \vee A^?_j$.

### 4.3.2 Дополнительные конструкции

Кроме случаев 1 и 2 рассмотрим еще одну




(3)

При этом зависимости $A_n$ от $A_i$ могут быть двух видов: $A_n$ работает, когда работают **все** вершины над ней, или хотя бы одна.

# 5 Реализация

В данный момент реализован [3] клиент к `collectd` серверам обрабатывающий поток рассылаемый ими в сеть. Разбор данных сделан относительно 4 версии демона. В качестве языка реализации был выбран Haskell и сервер HAppS, позволяющий в то же время предоставить удобный отладочный HTTP интерфейс. [4]

## 5.1 Внутреннее представление

### 5.1.1 Состояния узлов

Состояние узла [5] (или тип сообщения) в системе описывается структурой `StateV` со следующими вариантами:

[3] Код доступен в репозитории `git://grid.pp.ru/psha/hcollect` или через GitWeb `http://grid.pp.ru/git/?p=psha/hcollect/.git`

[4] Использована версия 0.8.8. Домашняя страница `http://happs.org`

[5] узлом называется не отдельная вычислительная система, а какая-то абстрактная единица про которую собирается информация

```
data StateV = SOk    -- узел в порядке
  | SBad             -- узел не в порядке, возможен сбой в ближайшее время
  | SFail            -- узел не работает
  | SNotAvail        -- узел недоступен
  | SUnknown         -- невозможно установить состояние
  deriving (Read, Show, Eq, Ord)
```

Состояние `SUnknown` используется, например, когда об узле слишком долго не поступало никакой информации. `SBad` — сигнализирует о возможном сбое, например, температура слишком высока. Состояния `SOk`, `SFail` и `SNotAvail` описаны в 4.1.

### 5.1.2 Сообщение

Сообщение в системе описывается следующей структурой

```
data Message = Msg{ mTarget :: HostID   -- узел, которому предназначено сообщение
  , mTime :: Integer                    -- временная метка
  , mState :: StateV                    -- декдоированое состояние
  , mSource :: String                   -- источник информации
  , mMisc :: Maybe String               -- какая-то дополнительная информация
  } deriving (Read, Show, Eq)
```

Кроме этого введен дополнительный класс для унифицированной обработки внешних сообщений и объявлена реализация для `Message`

```
class Convertable a where
  convert :: a → Message

instance Convertable Message where
  convert = id
```

### 5.1.3 Агрегация данных

Что бы можно было обнаруживать «колебания» и не тратя лишних ресурсов хранить какую-то историю изменений можно использовать метод, сходный с используемым в `rrd`.

```
data Aggr = Aggr{ aLog :: [Message]   -- обычная история, последние
  , aAggr :: [Message]                -- изменения состоянии, последние
  } deriving (Read, Show)
```

Функция обновления добавляет в `aLog` сообщение, обрезая список до какой-то фиксированной длины (сейчас используется константа 10). В `aAggr` головной элемент заменяется на новый, если он имеет то же состояние, или новый просто дописывается в начало. Таким образом, в `aAggr` хранятся 10 последних изменений состояния.

```
updateAggr :: Aggr → Message → Aggr
updateAggr (Aggr l a) m = Aggr (take 10 (m : l)) (take 10 (m : u a)) where
  u [] = []
  u (x : xs) = if mState x ≡ mState m then xs else x : xs
```

### 5.1.4 Узел

Узел, кроме зависимостей, описывается следующими полями

```
data Host = Host{ hName :: HostID          -- идентификатор узла
                , hState :: StateV         -- состояние узла
                , hCause :: Maybe Message  -- сообщение, ставшее причиной сбоя
                , hAggr :: Aggr            -- агрегированная история
                } deriving (Read, Show)
```

Поле `hCause`, в случае если узел находится в сбойном состоянии, содержит сообщение, ставшее причиной сбоя. `hMsg` используется для хранения нескольких последних сообщений для того, что бы можно было отслеживать кратковременные тенденции в изменении значений (возможно, имеет смысл хранить с агрегированием последних значений, что-то типа `rrd`).

### 5.1.5 Состояние системы

Состояние системы мониторинга описывается следующими компонентами

- список описаний узлов (используется `Data.Map.Map String Host`)

  ```
  type HostMap = M.Map String Host
  ```

- граф зависимостей (в данный момент не реализовано, для реализации разумно использовать `fgl`)

## 5.2 Обработка сообщений

В данный момент поддерживаются только сообщения от `collectd`. Сначала они преобразуются к внутреннему виду (см. 5.1.2), при этом состояние декодироуется только для сообщений типа `ping` и `hddtemp`. После приведения оно модифицирует состояние системы как описано ниже.

### 5.2.1 Изменение состояния системы

Все функции модификации состояния имеют тип $a \rightarrow \text{Message} \rightarrow a$ (выбран в соответствии с типом функции `foldl`). В коде определены следующие

- *update* :: *MonState* → *Message* → *MonState*

  Функция модификации всего состояния. В данный момент всего лишь вызывает обновлене списка узлов `updateHM`.

- *updateHM* :: *HostMap* → *Message* → *HostMap*

  Функция обновления списка узлов. Проверяет, есть ли узел которому предназначается сообщение в списке и, если такого нет, добавляет его. После этого вызывает `updateHost`.

- $updateHost :: Host \rightarrow Message \rightarrow Host$

Функция обновления узла. Пока только добавляет сообщение в список последних.

## 5.3 Возможная архитектура

Обработка сообщения может выглядеть следующим образом

- декодируется и преобразуется в родной для системы формат (5.1.2);
- проходит через фильтр и ненужные выкидываются;
- поступает на обработку и модифицирует состояние системы (5.2.1);
  - агрегируется, что-то типа `rrd` (5.1.3).
  - отправляется куда-нибудь в другую систему;

### 5.3.1 Обработка сообщений

$$\textbf{type } Filter = Message \rightarrow Bool$$
$$filterAllow :: Filter$$
$$filterDeny :: Filter$$
$$filterAllow \_ = True$$
$$filterDeny \_ = False$$
$$process :: (Convertable\ m) \Rightarrow Filter \rightarrow MonState \rightarrow m \rightarrow MonState$$
$$process\ f\ st\ m = \textbf{if } f\ m' \textbf{ then } update\ st\ m' \textbf{ else } st \textbf{ where}$$
$$m' = convert\ m$$
$$process' :: (Convertable\ m) \Rightarrow MonState \rightarrow m \rightarrow MonState$$
$$process' = process\ filterAllow$$

## 5.4 Проблемы

- преобразование данных из родного формата `collectd` (см. A) в данный момент производится только для плагинов `ping` и `hddtemp`.

# 6 Не рассмотренные проблемы

- замена дерева на ориентированный граф без циклов;
- приведение данных ко внутреннему виду должно проходить в два этапа
  - разбор записи (транспортный разбор);

– в зависимости от хоста решение, что означали данные (приведение к `StateV`);

# A Внешний источник `collectd`

В качестве внешнего источника данных используется `collectd`

## A.1 Внутреннее представление

### A.1.1 Разбор

Разбор данных производится с использованием `Data.Binary` по схеме, похожей на функцию `parse_packet` из `src/network.c` исходных кодов демона `collectd` версии 4.1.2.

### A.1.2 Сообщения `collectd`

В `collectd` есть структура `value_u`, содержащая либо целое значение счетчика (`int64_t`), либо дробное значение (`double`). В коде оно представляется типом `CVU`.

**data** *CVU* = *CVCounter Integer* | *CVGauge Double*
  **deriving** (*Show*)

Общая структура плагинов (каждое собранное демоном значение характеризуется 5 строками) и формат данных, передаваемых по сети, описывается типом `CollectValue`.

**data** *CollectValue* = *CV* { *cv_host* :: *String*
  , *cv_plugin* :: *String*
  , *cv_plugin_inst* :: *String*
  , *cv_type* :: *String*
  , *cv_type_inst* :: *String*
  , *cv_time* :: *Integer*
  , *cv_interval* :: *Integer*
  , *cv_values* :: [*CVU*]
  } **deriving** (*Show*)