



EMC® Watch4net®

Version 6.4u2

Backend Administration Guide

P/N: pending

Rev

EMC²

Copyright ©2005 - 2014 EMC Corporation. All rights reserved. Published in the USA.

Published October, 2014

EMC believes the information in this publication is accurate as of its publication date. The information is subject to change without notice.

The information in this publication is provided as is. EMC Corporation makes no representations or warranties of any kind with respect to the information in this publication, and specifically disclaims implied warranties of merchantability or fitness for a particular purpose. Use, copying, and distribution of any EMC software described in this publication requires an applicable software license.

EMC², EMC, and the EMC logo are registered trademarks or trademarks of EMC Corporation in the United States and other countries. All other trademarks used herein are the property of their respective owners.

For the most up-to-date regulatory document for your product line, go to EMC Online Support (<https://support.emc.com>).

Contents

Preface	iii
Document Organization	iii
Common Abbreviations and Acronyms	iii
Where to get help	v
1 Architectural Overview	1
1 The Collecting Component	2
1.1 The <i>APG</i> Data Model	2
1.2 Architecture	4
1.3 Data Collecting Adapter (<i>DCA</i>)	5
1.4 Data Collecting Filter (<i>DCF</i>)	5
1.5 Data Collecting Connector (<i>DCC</i>)	6
2 The Processing Component	7
2.1 Control Interface (<i>CI</i>)	8
2.2 Data Processing and Aggregation	8
2.3 Storage Connector (<i>SC</i>)	8
3 The Database Component	9
2 Configuration	11
1 The Collecting Components	11
1.1 General configuration	11
1.2 Running the Collecting Components as <code>root</code>	12
1.3 Subcomponents configuration	13
2 The Processing Component	17
2.1 Storage Connector configuration	18
2.2 Control Interfaces configuration	18
2.3 Storage Groups configuration	18
2.4 Maintenance tasks	23
3 The Database Component	24
3.1 MySQL	24

3.2 Oracle	25
4 The Management Interface	25
3 Deployment Cases	27
1 Single SMARTS / APG server	27
1.1 Introduction	27
1.2 Configuration	27
2 Distributed configuration	31
2.1 Introduction	31
2.2 Configuration	31
3 Duplicated Backend configuration	34
3.1 Introduction	34
3.2 Configuration	34
Appendix	36
A Raw value format	37
B Duration Notation	41
C Built-in Collecting subcomponents	43
1 Socket DCA	43
2 File DCC	44
3 Out DCC	44
4 Socket DCC	45
D Built-in Processing Subcomponents	47
1 Socket CI	47
2 Crossed Failover Socket CI	48
3 Telnet CI	49
4 MySQL Storage Connector	49
5 Oracle Storage Connector	51
E Backend webservice client	53

Preface

Document Organization

1. Architectural Overview	This section describes the Backend architecture and the role of each component
2. Configuration	This section describes the configuration of each component
3. Deployment Cases	This section describes common implementation scenarios

Common Abbreviations and Acronyms

<i>DCA</i>	Data Collecting Adapter
<i>DCF</i>	Data Collecting Filter
<i>DCC</i>	Data Collecting Connector
<i>CI</i>	Control Interface
<i>SC</i>	Storage Connector

Where to get help

EMC support, product, and licensing information can be obtained as follows:

PRODUCT INFORMATION - For documentation, release notes, software updates, or information about EMC products, go to EMC Online Support at:

<https://support.emc.com>

TECHNICAL SUPPORT - Go to EMC Online Support and click Service Center. You will see several options for contacting EMC Technical Support. Note that to open a service request, you must have a valid support agreement. Contact your EMC sales representative for details about obtaining a valid support agreement or with questions about your account.

1

Architectural Overview

The *APG* product consists in four major parts, each one having a specific role to play in the whole application:

- **APG Web Portal:** Customizable web user interface with advanced reporting capabilities.
- **Relational Database:** Database (built-in `MySQL` and `Oracle`) that stores raw and aggregated performance data.
- **Processing Component:** Aggregates and stores the collected data in an efficient way.
- **Collecting Component:** Collects, unifies and pre-processes data from any source.

This manual will focus on the **Processing** and **Collecting** components and will explain how to configure and use them. We will go through all the configuration files and the settings that are customizable inside them.

The overall architecture of *APG* and its interfaces with third-party applications is described on figure [1](#) below and illustrates how the components interact with each other. In this document, we will only address the *Collecting* and *Processing* components.

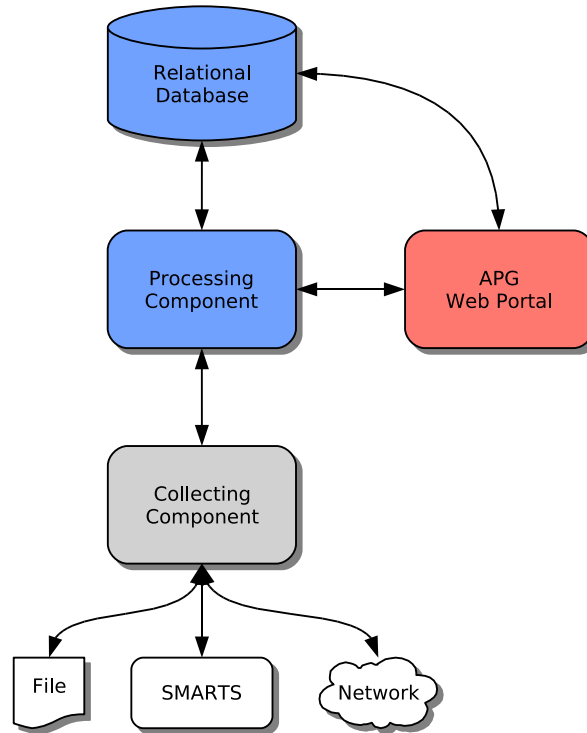


Figure 1: Global Architecture Overview

1 The Collecting Component

1.1 The APG Data Model

The fundamental element of *APG* data model is the raw value, which represents a floating-point value, at a specific date and for a specific indicator. A raw value can contain meta-data about the indicator. These meta-data are called properties.

For example, a raw value describing a `currentUtilization` may contain the name of the device it comes from and the `location` of that device.

Internally, *APG* has a flat data model: it does not have any knowledge about the relations between indicators. It only stores indicator properties.

On first look, it may appear as a drawback but actually, it is a major advantage since *APG* has no implicit hierarchy between data properties. It doesn't make any assumptions like "a device contains interfaces which contains several indicators". Then, *APG* can collect data from heterogeneous sources and still classify them in a convenient way in the Frontend.

Furthermore, thanks to this, users can define their own hierarchical data model, according to their needs.

Any *APG* collector we provide will automatically retrieve relevant data properties (e.g. indicator name, indicator owning device, indicator category, owning device category...). Besides of this, the user has the ability to customize these properties using several methods.

He may customize the property tagging process - which is done automatically when *APG* collects data -, or simply assign some user defined properties to any group of indicators, and then use them afterward to classify data in a more convenient way.

1.2 Architecture

All collecting components are managed by the Collector-Manager module.

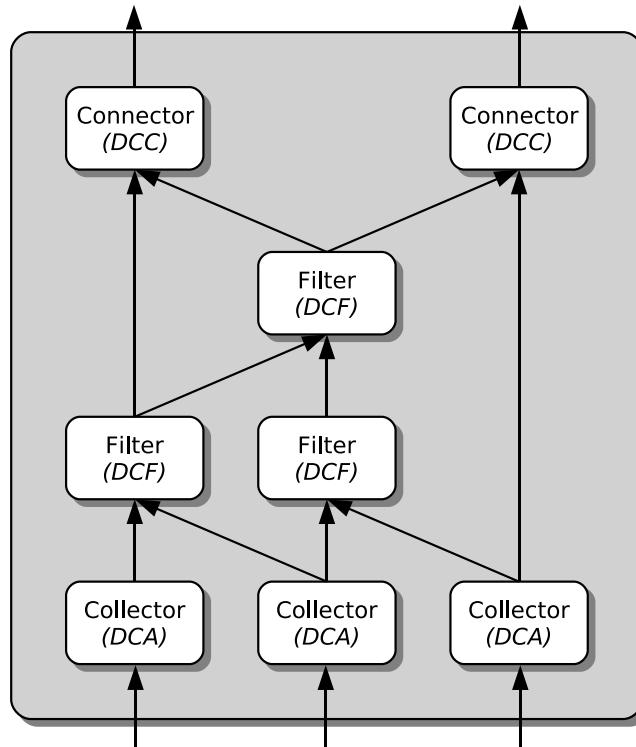


Figure 2: Collecting Component Architecture Overview

The data-collecting component of *APG* is a lightweight yet standalone application that can be distributed anywhere there are devices to be polled. This way, it can always be located near the data-source in order to optimize communication, data collection and transfers.

This application is involved in several tasks:

- it gathers performance data from any source and converts it to an unified raw value format,
- it discovers indicator properties (topology information, device properties, ...) and tags performance data with it,
- it pre-processes and filters performance data if necessary,
- it exports pre-processed performance data to the following components in the chain.

To achieve these tasks, data flow goes through a fully customizable oriented graph of pluggable sub-components. This graph is described in a simple XML file explained in [section 2](#), on [page 11](#).

1.3 Data Collecting Adapter (*DCA*)

The adapter is the component that will convert any data from any source into a standardized raw value format. It is also responsible for discovering, managing and tagging indicator properties.

These kinds of components can do any sort of data processing or filtering that is directly related to data source specific behavior or requirements. Source independent processing should be done with filters.

DCA modules include the `Smarts-Collector`, the `Text-Collector`, the `SNMP-Collector`, etc.

1.4 Data Collecting Filter (*DCF*)

A filter is a subcomponent that is placed between adapters and connectors. Thus, it receives data from adapters or other filters and may or may not send it to other filters or connectors.

Filters can modify the data flow, altering data or meta-data, filtering out, duplicating or aggregating some values.

Another interesting usage of a filter is an adaptive pre-caching processing if the data pipe is temporarily broken. This is what the `failover` filter does, for example.

DCF modules include the `Fail-Over Filter`, the `Property-Tagging-Filter`, the `Simple-Inline-Calculation-Filter`, etc.

1.5 Data Collecting Connector (DCC)

The connector is the last element of a data path. Usually, it is used to export the collected data to another *APG* component or any other data sink. A typical example is a connector which will send data to the data processing component, using plain text over a socket, in a reliable way.

This subcomponent is also responsible for the communication channel integrity: by example, the *Socket Connector* which sends data through a *TCP/IP* socket is responsible for keeping the connection alive and has to warn the whole data pipe if an error occurs.

DCC modules include the *Cache-Connector*, the *File-Connector*, the *Socket-Connector*, etc.

2 The Processing Component

As explained before, data is collected from all the available sources. Then, these data are transmitted to an aggregation system that deals with calculation and retention. Actually, the received data are only raw ones.

This place is where simple but useful kinds of aggregates are computed from the raw values, such as the average value, minimum, maximum, sum, etc... Finally, data are stored on a persistent storage system, typically a database.

So the Aggregation Component is made of three parts. The main one only processes the data, while the two others connect it to the outside and to the persistent storage system.

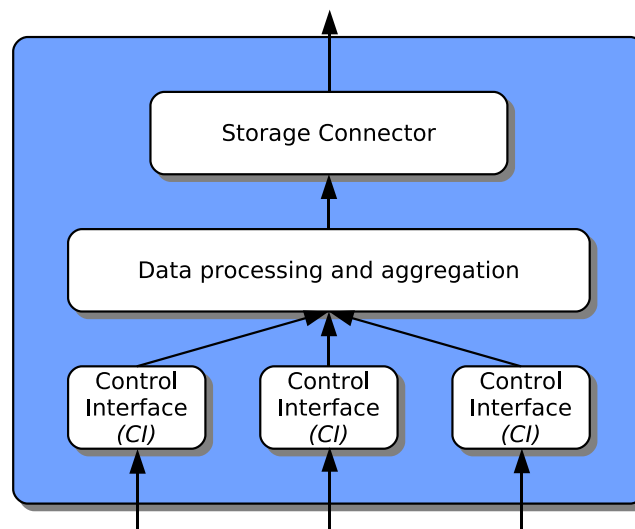


Figure 3: Collecting Processing Architecture Overview

2.1 Control Interface (CI)

The control interface offers a flexible means to control the aggregation component. It can receive values or orders from any external component and manage the aggregation subcomponent accordingly.

This enables the end-user to have his own collecting component to interact with *APG* core and the administrator to control the process.

2.2 Data Processing and Aggregation

This subcomponent receives raw values and computes aggregates. It is also responsible for data integrity and fault tolerance management.

It creates blocks of data directly stored into files (in order to avoid data loss in case of failure) and updates in-memory aggregates (which are also periodically flushed to hard disk). This subcomponent manages round-robin rotations, telling the *Storage Connector* to remove outdated blocks and write new ones, so that data is automatically removed when expired.

2.3 Storage Connector (SC)

The storage connector is the pluggable component that manages dialog with the database (or any other storage system). It is driven by asynchronous orders delivered by the aggregation component.

This part is also responsible for data integrity since it must not delete aggregation component files unless they have been successfully written to storage system.

It means that if the data insertion into the database fails, the data file will remain undeleted inside the temporary data directory.

Note: The database link remains critical since the processing component needs to access in-database data frequently (at least with lazy data loading policy) and write huge amounts of data regularly.

However, some recovery mechanisms avoid complex maintenance and in the worst situation, a simple restart of this component will solve any issue.

3 The Database Component

The *Processing Component* maintains an open database schema which can be accessed by third-party applications (especially the *APG Web Portal*). This schema is exposed on figure 4 page 10.

The meta-data tables contain the variables and their properties. There is a 1-n relation between the table *data_variable* and the table *data_property*.

The *data_variable* table contains the variable id (i.e. the internal variable identifier), the variable name (i.e. the external unique identifier for the variable), and the storage group (the retention group the variable belongs to, as defined in section 2 on page 17).

The *data_property* contains the properties associated to a variable. Each property is identified by a name and a value.

The performance tables contain the performance data for every stored aggregate and for raw values. The raw values are stored in a specific table called *data_0* which contains the internal variable identifier, the timestamp and the corresponding value. The raw values timestamps are unpredictable since they directly depend on the data sources polling rate.

The aggregated values are stored in tables called *data_<aggregate period, in seconds>* (for example, on the figure 4, we can see the table *data_3600* for hourly aggregates). These tables contain the internal value identifier, the aggregate timestamp (which is always predictable since it is a multiple of the aggregate period), the average value, the minimum raw value, the maximum raw value, the sum, the last raw value, the number of raw values used to calculate these aggregates and the timestamp of the last raw value.

The performance data tables may differ according to your aggregate and retention settings (please refer to section 2 on page 17).

Database component modules include the MySQL, the APG-Datastore, etc.

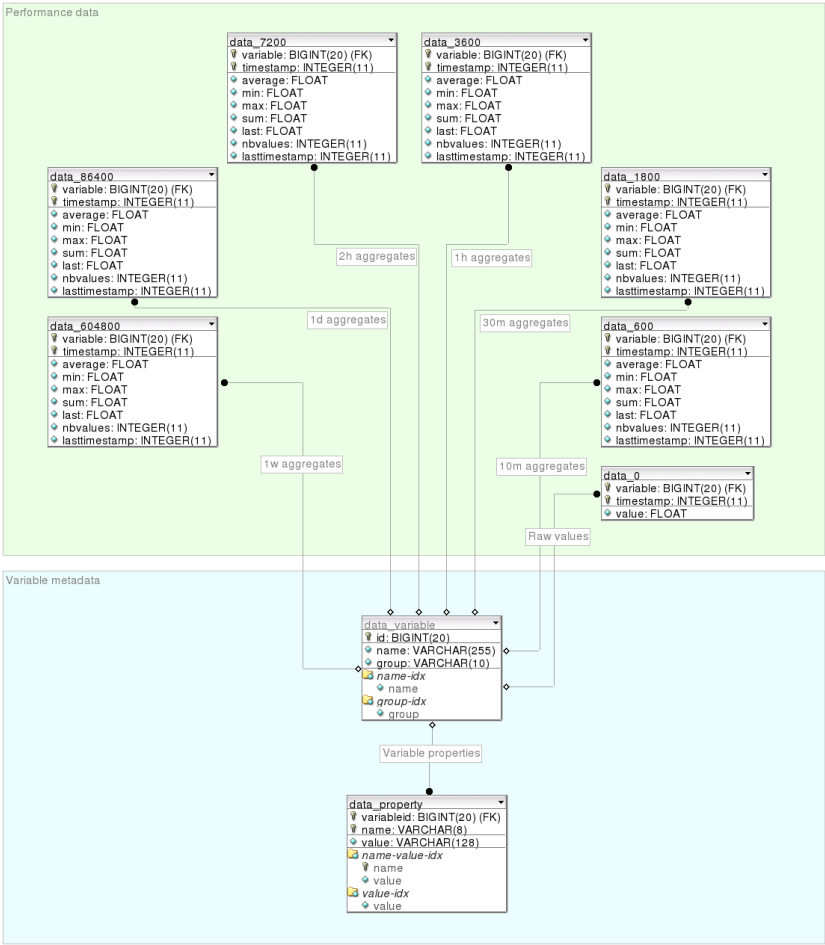


Figure 4: External Database Schema

2

Configuration

1 The Collecting Components

1.1 General configuration

Each collecting component is configured through a single *XML* file which describes the graph of plug-gable components illustrated on figure [2](#) on page [4](#). This file contains the name and configuration of every component and their link.

This *XML* file is located in the `conf` directory of the `Collector-Manager` module. The following example illustrates the global organization of this file.

```
<?xml version="1.0" encoding="UTF-8"?>

<config xmlns="http://www.watch4net.com/APG/Collecting"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.watch4net.com/APG/Collecting_collecting.xsd">

  <connectors>
    <!-- connectors will be declared in this section -->
  </connectors>
  <filters>
    <!-- filters will be declared in this section -->
  </filters>
  <collectors>
    <!-- collectors will be declared in this section -->
  </collectors>

</config>
```

1.2 Running the Collecting Components as root

Some collectors may require administrative rights in order to perform specific tasks. For instance, the ICMP collector needs to access raw sockets in order to perform PING requests. As a consequence, on Unix systems, running with `root` privileges is the easiest way to grant the required permissions.

There are some best practices to keep in mind before setting up collecting components to have `root` privileges:

- Do not run all collectors with `root` privileges. There is only a very small subset of them that need `root` privileges. First, identify the relevant collectors (this should be highlighted in their respective documentation), install another instance of the Collector-Manager, and register these collectors in that new instance.
- Only register the minimal subset of filters in this new Collector-Manager instance. The less components in the Collector-Manager instance running with `root` privileges, the better.

Once the new collecting component architecture is ready, open the service configuration file (located in the new Collector-Manager instance's `conf` directory and named `unix-service.properties`) and apply the following modifications:

- In Main Service Properties, uncomment the line `apg.user=root`

- In Additional Java Parameters, uncomment the two lines that add the properties named `java.rmi.server.hostname` and `com.sun.management.config.file`.

Then reinstall this Collector-Manager service instance using the Module-Manager, and then finally launch it.

1.3 Subcomponents configuration

Each collecting subcomponent will be declared the same way, using an *XML* element called `connector` (for Data Collecting Connector), `filter` (for Data Collecting Filter) or `collector` (for Data Collecting Adapter). These *XML* elements have 3 attributes:

- `enabled` (not required, default is *true*) is a convenient way to disable a subcomponent without having to remove it from the configuration file. If the disabled subcomponent appears in the `next` attribute of any other subcomponent, it will be silently ignored. Possible values are *true*, *false* and *skip* (only available for filters). A filter on a *false* state will stop the flow of data while a skipped filter will behave like a pass-through.
- `name` (required) is the subcomponent name which will identify it in the collecting graph. Therefore, it must be unique for each collecting component.
- `next` (required only for `collector` and `filter`, forbidden for `connector`) tells the subcomponents which ones are its next elements. If the subcomponent has data to send, it will send it to each subcomponent listed in the `next` attribute (space separated list).
- `conf` (required) is a configuration file name which will be passed to the subcomponent. Note that the path is relative to the collecting component working directory.

Note: The `conf` attribute may not link to a real directory on the file system. It happens when a module contains multiple components. Please, refer to the component documentation for more details.

Note: The previous way of configuring a component with a sub tag *classname* and *config-file* is still working but will generate a warning in the log files.

The following *XML* fragment declares a `collector` called *Collector1* which will send data to *Filter1* and which is enabled.

```
<collectors>
  <collector enabled="true" name="Collector1" next="Filter1" conf="myconf.conf" />
</collectors>
```

You can define as many collectors, filters and/or connectors as you need, as long as they stay in their corresponding regrouping elements.

Here is a complete example of a collecting configuration file. The resulting collecting graph is shown on figure 5. It show the path the data follows, from the data source to the processing component.

```
<?xml version="1.0" encoding="UTF-8"?>

<config xmlns="http://www.watch4net.com/APG/Collecting"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.watch4net.com/APG/Collecting_collecting.xsd">

  <connectors>
    <connector name="Connector1" ... />
    <connector name="Connector2" ... />
    <connector name="Connector3" ... />
  </connectors>

  <filters>
    <filter name="Filter1" next="Connector2" ... />
    <filter name="Filter2" next="Connector3" ... />
  </filters>

  <collectors>
    <collector name="Collector1" next="Filter1_Connector1" ... />
    <collector name="Collector2" next="Filter1_Filter2" ... />
  </collectors>

</config>
```

The configuration of each kind of collecting subcomponent is described either in its own manual either in this particular manual appendix (on page 43) for built-in subcomponents.

APG also comes with a specific *EMC² Smarts InCharge* Collector whose features and configuration are explained in the *APG-Smarts-Collector-Admin* document.

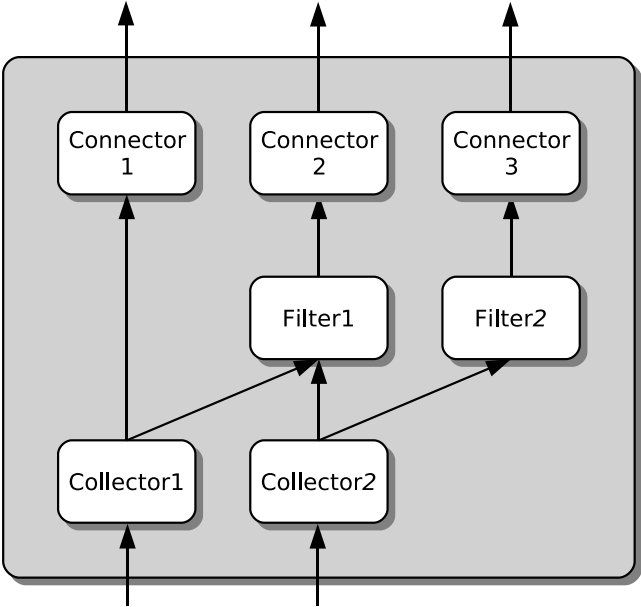


Figure 5: Collecting graph example

2 The Processing Component

The processing component is the heart of the whole *APG-Backend* module. It controls the way data are received, processed and stored before being used. Its configuration is defined in an *XML* file (usually called *config.xml*), and some of its parts require a specific attention.

```
<?xml version="1.0" encoding="UTF-8"?>
<backend xmlns="http://www.watch4net.com/APG/Backend" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:schemaLocation="http://www.watch4net.com/APG/Backend_config.xsd">

    <persistent-storage ...>
        <!-- ... -->
    </persistent-storage>

    <control-interfaces>
        <!-- ... -->
    </control-interfaces>

    <temporary-files ...>tmp</temporary-files>

    <groups>
        <!-- ... -->
        <aggregates-definition config="conf/aggregates.xml" />
    </groups>

    <maintenance-tasks>
        <!-- ... -->
    </maintenance-tasks>

</backend>
```

That main configuration file can be divided in 5 distinct parts:

- the `persistent-storage` that setups the storage connector subcomponent,
- the `control-interfaces` where you will declare the available control interfaces,
- the `temporary-files` contains the location where the processing component will write its temporary data files. This directory is relative to the processing component installation directory. For best performance, it is recommended to set this path on a fast drive, and as near as possible from the database.
- the `groups` element is where storage groups will be declared. A storage group consists of a specific retention profile that will be applied to variables belonging to this group. The nested

`aggregates-definition` element is the name of the aggregate descriptor file that will be explained in section 2.3.

- `maintenance-tasks` is where automated maintenance tasks are declared and configured.

Note: It is also recommended to set the `temporary-files` location on a separate partition: although that directory should remain quite empty most of the time, if ever the database link goes down it might be filled with temporary values. Thus having it on a temporary partition will prevent filling the Backend and/or database partition with these data.

2.1 Storage Connector configuration

A processing component can only have one *Storage Connector* which is responsible for storing data persistently. The *Storage Connector* is declared like collecting subcomponents:

```
<persistent-storage name="Storage" type="StorageConnector" config="conf/the-conf.xml" />
```

The `name` attribute is not important there, it is only used for logging. The configuration file is specific to each *Storage Connector*. The configuration for built-in connectors will be described in appendix D on page 47.

2.2 Control Interfaces configuration

The control interfaces are declared in the `control-interfaces` block. Each control interface must be declared with a nested `control-interface` block like the example below.

```
<control-interface name="CI" type="InterfaceAdapter" config="conf/the-conf.xml" />
```

The configuration file is specific to each *Control Interface*. The configuration for built-in control interfaces will be described in appendix D on page 47.

2.3 Storage Groups configuration

The storage group configuration is the most important part of the processing component configuration. This configuration involves two *XML* files:

- the main configuration file, `config.xml`, located at `<APG>/Backends/APG-Backend/<instance name>/conf`
- the aggregate descriptor file, `aggregates.xml`, located at `<APG>/Backends/APG-Backend/<instance name>/conf`

The aggregates are defined in the `aggregates.xml` file. These are the ones that will be available for reporting in the *Frontend* for all the data that come from that Backend.

The properties of an aggregate in the `aggregates.xml` file **HAVE TO** be set **ONCE AT THE BEGINNING** and **NEVER** modified after the beginning of the processing. In fact, failing to do so may (often will) result in losses and corruption for the already collected data for that aggregate.

You could, however, add new aggregates, which will be computed from that time forwards, or remove an aggregate. If you remove an aggregate, the current data for that aggregate will remain available in the Frontend, until and unless the data is removed from the database, but no new data for the aggregate will be recorded.

Let's examine the standard aggregate descriptor file, `aggregates.xml`, supplied with *APG*:

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- ***** -->
<!-- * If you don't want to RUIN ALL YOUR DATA you should NOT TOUCH THIS FILE. * -->
<!-- ***** -->

<aggregates xmlns="http://www.watch4net.com/APG/Backend" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:schemaLocation="http://www.watch4net.com/APG/Backend/aggregates.xsd">

    <raw-data split="6h" />
    <aggregate name="10min" period="10m" split="6h" />
    <aggregate name="1hour" period="1h" split="36h" />
    <aggregate name="1day" period="1d" split="36d" />
    <!-- <aggregate name="1dayAligned" xsi:type="AlignedAggregate" property="tz" period="1d"
split="36d" /> -->
    <aggregate name="1week" period="7d" split="252d" />
    <!-- <aggregate name="1weekAligned" xsi:type="AlignedAggregate" property="tz" period="7d"
split="1252d" /> -->

</aggregates>
```

This configuration file uses the [Duration Notation](#) defined in appendix B page 41.

For each aggregate, except real-time values that are not aggregates, the period of that aggregate has to be specified, along with a symbolic identifier called `name` that will be used in the main configuration file. The `split`¹ attribute represents the period which will be used to partition tables in the database. The split mechanism is an internal one, but it has an impact on the effective storage size.

An aggregate can be declared as an `AlignedAggregate`. An `Aligned Aggregate` processes values based on their time zone and produces time zone independent aggregated values. For example, for a given metric, a daily `Aligned Aggregate` would aggregate values for the real day, using the local time, of its declared time zone, instead of the `UTC` day.

This way, for example, you can have metrics from different locations and time zones, aligned into a graph that expresses all values in their local time zone. The graph would represent an (imaginary) universal day, so you can, for example, compare peaks and troughs according to when they occurred locally, and from the users perspective in each location. An occurrence happening at 9:00 in one location/time zone, would be aligned with an occurrence at 9:00 in another location/time zone.

Time-shifted aligned aggregates are only available for raw data in timeseries.

Without using the time-shifted aggregates, information is expressed from the vantage point of the report time zone, so values with timestamps from other time zones would be shifted in the day according to how their time zone is offset from the local time. For example, data from 9:00 America/Montreal (GMT -5:00) time, would be aligned with data from 14:00 Europe/London (GMT +0:00) time.

To use `1dayAligned` and/or `1weekAligned` aggregates, you have to uncomment the corresponding lines in the `aggregates.xml` file, and the `config.xml` file, displayed below. As well, you have to uncomment the following lines to give access to the timeseries database with aligned aggregates, in the `APG.xml` file, located at: `<APG>/Web-Servers/Tomcat/<instance name>/conf/Catalina/localhost`

```
<!-- Gives access to the timeseries database. -->
<!-- <ResourceLink name="jdbc/APG-DB" global="jdbc/APG-DB" type="javax.sql.DataSource" />
-->

<!-- Gives access to the timeseries database with aligned aggregates. -->
```

¹The `APG` size calculator takes care of these settings, so be sure to modify them as well in that tool if they have been changed here.

```
<ResourceLink name="jdbc/APG-DB:{_timezones=tz:America%2FMontreal_}" global="jdbc/APG-DB"
type="javax.sql.DataSource" />
```

Be aware of the following about the `APG.xml` file:

- You must only uncomment one timeseries database, so if you uncomment the timeseries database with aligned aggregates, the other timeseries database must be commented out, just as indicated in the code above.
- If you uncomment the timeseries database with aligned aggregates, the Frontend will create aligned aggregates for 1-hour periods and less, on a best effort basis, even if daily and weekly aligned aggregates are not configured.
- For the `timezones=tz:America%2FMontreal` entry, the time zone chosen must match that set in the `config.xml` file (see below). As well, the “tz” property could be different, if you used a different value for the `property` attribute for the aggregate tags in the `aggregates.xml` file. Notice that where a / should be between America and Montreal in the time zone element must be replaced by %2F to make it work.

By default, each raw value can indicate which time zone it belongs to using the `tz` property. You can add the `tz` property using the Property Tagging Filter, for example. You should use a time zone expressed in the Olson time zone database format (also referred to as the `tz` database) such as `America/New_York` or `Europe/Paris`, which takes into account such things as daylight savings time. You can search for `tz` database or Olson `tz` database sources online to find the proper syntax for the time zones you want to use.

Note: If you cannot use a property called “tz” for the time zone information, because for example, there is another property called `tz` coming from a particular technology, you can define another property to hold the time zone information. This is done per aggregate in its `property` attribute in the `aggregates.xml` file.

Aligned `tz` aggregates for daily and weekly time periods are stored in the Backend (by default, and if included as described in the configuration files in this section). The Frontend will compute aligned aggregates on the fly on a best effort basis, for all aggregates of 1 hour or less. For more information, see the *APG-Frontend-User-Guide*. If the time zone information is missing, the default defined in the storage group will be used (see the next paragraphs).

If data is relegated to a single time zone, you do not need to indicate a `tz` property for the raw data.

Using an aligned aggregate in this case in the Frontend will simply shift daily and weekly time periods to be aligned with the local time zone. Otherwise days and weeks will be delineated according to UTC time.

Once the aggregates are configured, retention periods need to be defined for each of them in the main `config.xml` file. This step is to be performed for each particular storage group. For each known group, a group element needs to be declared, nested into the `groups` element of the `config.xml` file.

```
<group name="group">

  <raw-data span-time="31d" />
  <aggregate name="10min" span-time="31d" />
  <aggregate name="1hour" span-time="61d" />
  <aggregate name="1day" span-time="365d" />
  <!-- <aggregate name="1dayAligned" xsi:type="AlignedAggregate" default-timezone="America/
Montreal" span-time="365d" /> -->
  <aggregate name="1week" span-time="2555d" />
  <!-- <aggregate name="1weekAligned" xsi:type="AlignedAggregate" default-timezone="America/
Montreal" span-time="2555d" /> -->

  <consistency-checks strict-ordering="true" maximum-time-drift="1d" />
  <variable-removal last-update-retrieval-aggregate="1week" idle-timeout="365d" />

</group>
```

The following is more information useful to configuring the `config.xml` file properly:

The `config.xml` file uses the [Duration Notation](#) defined in [appendix B](#) page 41.

In a group descriptor, you will have to specify the retention time, for every aggregate defined in the aggregate descriptor file (the `span-time` element). You can also disable some aggregates by using 0 as `span-time` element. A disabled aggregate is completely ignored, never computed and never stored.

For *Aligned Aggregates*, you have to use the `AlignedAggregate` type. It allows you to define a default time zone which will be used when raw values are missing the time zone property. When omitted, it uses the default system time zone.

`consistency-checks` and `variable-removal` elements deal with data consistency checks and variable removal.

The `strict-ordering` attribute is used to ensure all raw values are received with increasing timestamps, thus preventing the processing of old data. The `maximum-time-drift` attribute specifies the maximum time drift allowed to raw values. It's used when the raw value timestamp is ahead of the Backend timestamp. The raw value will be discarded if its timestamp is too much ahead more than max time drift of the Backend timestamp.

The `variable-removal` element is used to define how the variable last update time will be retrieved at startup (`last-update-retrieval-aggregate`) and how long a variable can remain idle (i.e. without new data for it), based on the last update time if found or the Backend uptime if not found, before being definitely removed (`idle-timeout`).

2.4 Maintenance tasks

Finally, the global maintenance tasks of the processing component have to be defined. Here are the settings that drive such tasks:

- the period of the memory flushing task (in-memory data is written to disk to prevent data loss on failure),
- the period of the automatic commit task (writes temporary files to the database),
- the period of the clean-up task (removes outdated data from the database).

The XML part to look at is located at the end of the main configuration file.

```
<maintenance-tasks>

  <flush period="30m" aggregate-period-threshold="30m" />

  <commit period="4m" file-age-threshold="2m" />

  <clean-up period="1h" />

</maintenance-tasks>
```

These configuration elements use the [Duration Notation](#) defined in appendix B page 41.

Their meaning is not that simple but is explained below:

- the `flush` element contains the period of the flush task. The `aggregate-period-threshold` attribute is the last aggregate that will not be flushed. Any aggregate whose period is bigger than `aggregate-period-threshold` will be flushed by the flush task.

- the `commit` element contains the period of the commit task and an additional parameter. The `file-age-threshold` attribute is the maximum acceptable age for a file. It means that a file younger than `file-age-threshold` will not be committed by the commit task.
- the `clean-up` attribute contains the period of the clean-up task. It can also contain `time-reference` attribute to define which timestamp to use: last timestamp of each group (`group-last`) which is the default or the current time (`current`).

3 The Database Component

The database component is the heart of *APG*, and both the Backend and the Frontend are not able to run without a properly configured database.

By default, *APG* comes with a MySQL database, well configured for most of the cases. A couple of settings can still be modified in order to best fit the server capabilities (in terms of memory essentially).

There is also another adapter provided for an Oracle database. However, in that case, the database has to be up and running before *APG* is installed.

Note: It is strongly discouraged to execute a MySQL dump (`mysqldump` command) on large databases (>10GB). The `mysqldump` command can run for several hours (if not days) on large databases. The backup of database raw files is a much better option with less overhead.

3.1 MySQL

APG has been designed to run using a MySQL database. That is why that software is provided during the installation (*MySQL* and *MySQL-Maintenance-Tool* modules), and already has some adequate settings.

In order to get the maximum from that database, some settings will still require attention.

First part is the link from the processing component of the Backend to the database itself. In order to achieve this, the *MySQL Storage Connector* has to be used as Storage Connector in the main configuration file.

The *MySQL Storage Connector* is the database manager specialized for *MySQL* database. It uses optimized procedures in order to get the best of a *MySQL* database.

To use this storage connector, simply declare it in the processing component configuration file as shown in the following example:

```
<persistent-storage name="MySQL" type="MySQLStorageConnector" config="conf/mysql.xml" />
```

3.2 Oracle

APG needs at least the **Oracle 10.1 Enterprise Edition** (10.2 and 11.2 are supported also) and some basic configuration, apart from the scripts run at installation. To use this storage connector, simply declare it in the processing component configuration file as shown in the following example:

```
<persistent-storage name="Oracle" type="OracleStorageConnector" config="conf/oracle.xml" />
```

For more information, see the Oracle administration documentation.

4 The Management Interface

The management interface allows the Collector-Manager to communicate some information to its collecting subcomponents. Some of the collecting subcomponents have a GUI (like the *transaction-collector*) that allows remote configuration. The communication between those GUI's and the Collector-Manager is done via this interface.

Collecting subcomponents that have their own interface use it to configure their instances remotely—each type with their own set of methods. The role of the Collector-Manager's interface is only to communicate which collecting subcomponents have their own management interface with which they can be managed.

By default, the Collector-Manager interface will not start when the Collector-Manager is started. Some changes need to be performed on the default configuration file `web-service.xml`. This file is located at: `<APG>/Collecting/Collector-Manager/Default/conf`

```
<?xml version="1.0" encoding="UTF-8"?>

<config xmlns="http://www.watch4net.com/APG/Configurator/WebService"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.watch4net.com/APG/Configurator/WebService_web-service.xsd">
```

```
<web-service management="false">
  <url>http://localhost:40015/</url>
  <login>my-login</login>
  <password>my-password</password>
</web-service>

</config>
```

The `management` attribute determines if the management interface is used or not. The `url` tag determines the port to use for the Collector Manager interface. The `login` and `password` tag determines the credentials to be used, which the collecting subcomponents will use to access the Collector-Manager's interface.

Deployment Cases

1 Single *SMARTS* / *APG* server

1.1 Introduction

Even if *APG* has been designed to fit distributed environment, it is sometimes interesting to have the whole suite on a single server. It is useful when *APG* is used on a medium size environment where it is impossible or useless to have dedicated servers for each component.

1.2 Configuration

You can design any collecting graph, but you have to use the *Cache Connector*. Any collecting sub-component that should send data to the processing component should have directly or indirectly the *Cache Connector* as next element.

The following *XML* file is the collection configuration to reproduce the collecting graph represented on figure 6 page 29.

```
<?xml version="1.0" encoding="UTF-8"?>

<config xmlns="http://www.watch4net.com/APG/Collecting"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.watch4net.com/APG/Collecting_collecting.xsd">

  <connectors>
    <connector enabled="true" name="Backend"
      conf="Collector-Manager/Default/Cache-Connector/conf/cacheconnector.xml" />
  </connectors>

  <collectors>
    <collector name="SMARTS-1" next="Backend"
      conf="Smarts-Collector/Default/conf/smarts1.xml" />
    <collector name="SMARTS-2" next="Backend"
      conf="Smarts-Collector/Default/conf/smarts2.xml" />
  </collectors>

</config>
```

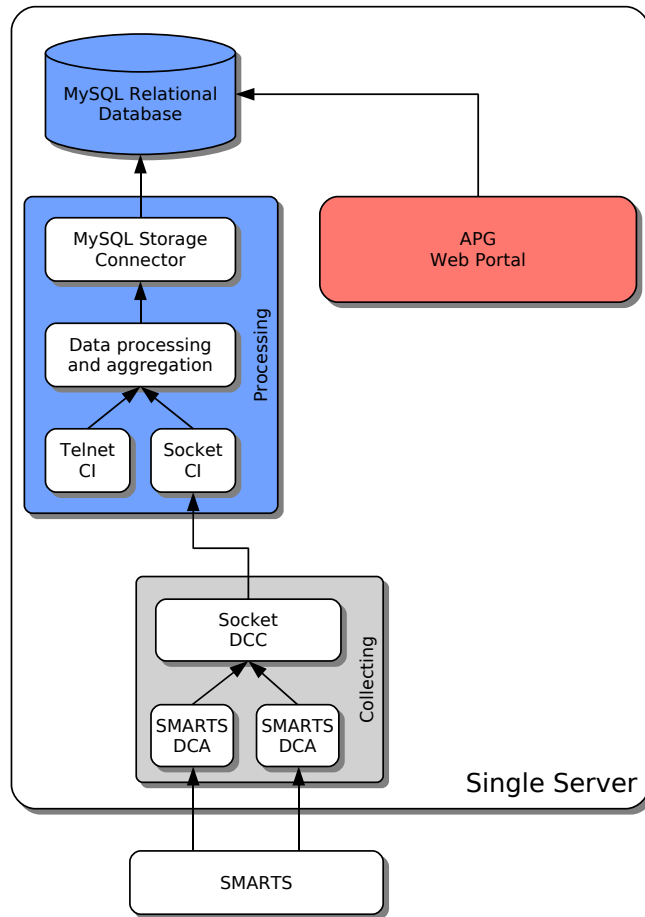


Figure 6: Single server architecture

The processing component has to use the *Socket Control Interface* or the *Crossed Failover Socket Control Interface*, but not the two at the same time. The *Socket DCC* of any collecting component will have to be set to the right host/port setup in the *Socket Control Interface* (or in the *Crossed Failover Socket Control Interface*) configuration file. Here is the relevant part of the configuration file for this kind of deployment scenario.

```
<persistent-storage name="MySQL" type="MySQLStorageConnector" config="conf/mysql.xml" />

<control-interfaces>
  <control-interface name="Socket-Collector" type="SocketCollectorInterface" config="conf/
socketinterface.xml" />
  <!--<control-interface name="Crossed-Failover-Socket-Collector" type="
CrossedFailoverSocketCollectorInterface" config="conf/crossedfailoversocketinterface.xml" />-->
  <control-interface name="Telnet-Control" type="TelnetControlInterface" config="conf/
telnetinterface.xml" />
</control-interfaces>
```

2 Distributed configuration

2.1 Introduction

This is the typical deployment scenario where data sources are distributed over a wide environment and where Backend components have to face important data flows. The figure [7](#) on page [32](#) illustrates this deployment scenario. Processing component, database and web portal may be hosted by different server. However, for optimal performances, they should be located in the same low latency network.

2.2 Configuration

For distributed environment, collecting components rely on the *Socket DCC* to send collected data to the processing component over an IP network. Any collecting graph can be built.

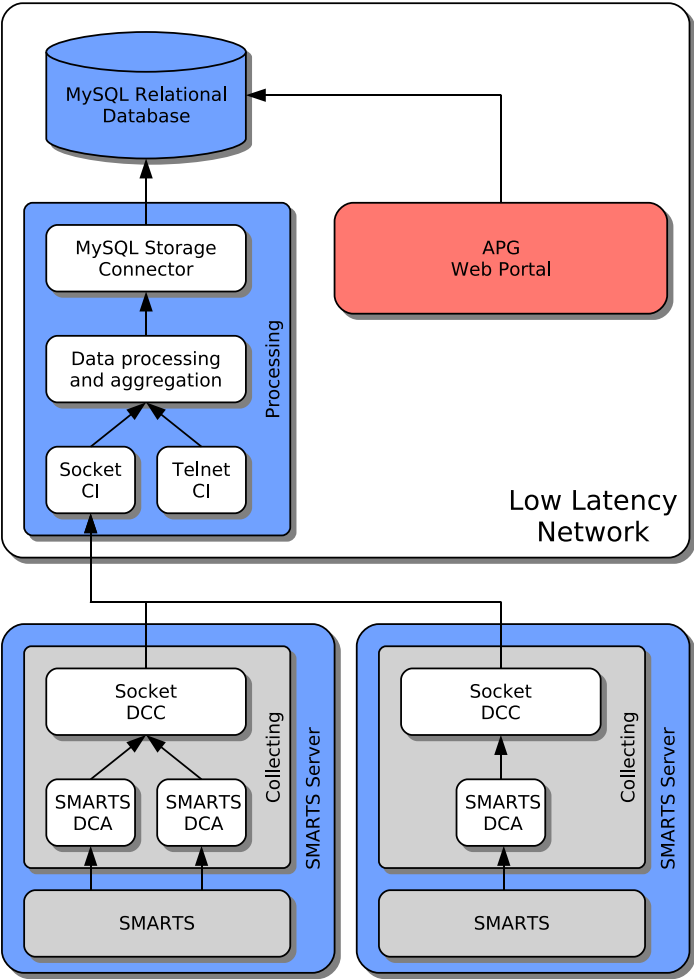


Figure 7: Distributed architecture

```
<?xml version="1.0" encoding="UTF-8"?>

<config xmlns="http://www.watch4net.com/APG/Collecting"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.watch4net.com/APG/Collecting_collecting.xsd">

  <connectors>
    <connector enabled="true" name="Socket"
      conf="Collector-Manager/Default/Cache-Connector/conf/cacheconnector.xml" />
  </connectors>

  <collectors>
    <collector name="SMARTS-1" next="Socket"
      conf="Smarts-Collector/Default/conf/smarts1.xml" />
    <collector name="SMARTS-2" next="Socket"
      conf="Smarts-Collector/Default/conf/smarts2.xml" />
  </collectors>

</config>
```

The collecting component will receive data from the *Socket Control Interface* or from the *Crossed Failover Socket Control Interface*, but not the two at the same time. The *Socket DCC* of any collecting component will have to be set to the right host/port setup in the *Socket Control Interface* (or in the *Crossed Failover Socket Control Interface*) configuration file. Here is the relevant part of the configuration file for this kind of deployment scenario.

```
<persistent-storage name="MySQL" type="MySQLStorageConnector" config="conf/mysql.xml" />

<control-interfaces>
  <control-interface name="Socket-Collector" type="SocketCollectorInterface" config="conf/
socketinterface.xml" />
  <!--<control-interface name="Crossed-Failover-Socket-Collector" type="
CrossedFailoverSocketCollectorInterface" config="conf/crossedfailoversocketinterface.xml" />-->
  <control-interface name="Telnet-Control" type="TelnetControlInterface" config="conf/
telnetinterface.xml" />
</control-interfaces>
```

3 Duplicated Backend configuration

3.1 Introduction

This scenario is based on the previous one but is suitable for high availability requirements or very specific needs. Indeed, duplicating the Backend is useful to create and maintain two distinct databases eventually with different retention profiles. The secondary database can be used as a fail-over database for high availability or for data archiving (with a different retention profile). In both cases, the web portal can be configured to use several databases.

3.2 Configuration

The configuration is not really different from the previous scenario. The main difference is that data will be duplicated at some point. For this example, data is duplicated inside the collectors, which will increase the traffic between the *APG* network and remote collectors. Another approach is to use the *Socket DCA* to create an intermediate collector inside the *APG* network which will be used to duplicate raw data. To duplicate data, it simply requires to setup more than one `next` subcomponent inside a collector. For example, the collector of the figure 8 would be configured like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="http://www.watch4net.com/APG/Collecting"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.watch4net.com/APG/Collecting_collecting.xsd">

  <connectors>
    <connector enabled="true" name="Socket-1"
      conf="Collector-Manager/Default/Cache-Connector/conf/cacheconnector1.xml" />
    <connector enabled="true" name="Socket-2"
      conf="Collector-Manager/Default/Cache-Connector/conf/cacheconnector2.xml" />
  </connectors>

  <collectors>
    <collector name="SMARTS" next="Socket-1_Socket-2"
      conf="Smarts-Collector/Default/conf/smarts.xml" />
    <collector name="SMARTS-2" next="Socket"
      conf="Smarts-Collector/Default/conf/smarts2.xml" />
  </collectors>
</config>
```

The processing component configuration is not modified from the previous example.

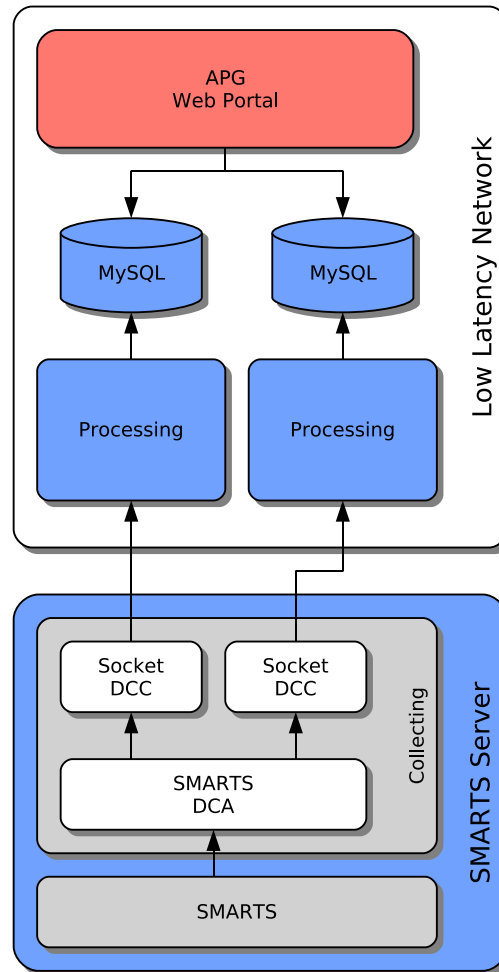
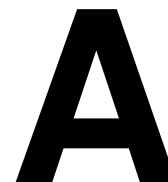


Figure 8: Dual Backend architecture



Raw value format

A raw value has several fields:

- *timestamp* is the value date expressed as a standard Unix timestamp (number of seconds since epoch) based on UTC. If this value is negative, the raw value is called *uninitialized* and its value will be ignored.
- *groupname* is the group of the variable. This group is used to distinguish which retention configuration to apply to this value.
- *variable* is the globally unique variable identifier.
- (*property,value*) are the property couples which decorate this variable. These properties are stored in database when the variable is created, thus it is not necessary to send them for every raw value. However some filters use the raw value properties. Not sending the raw value properties may cause incorrect data to appear in the database.
- *action* is a special (optional) flag to carry some action to be done by the application which will process the raw value (typically the Processing Component). A raw value can only contain *ONE* flag. There are currently two defined flags:
 - *FORCE_REFRESH_ACTION* tells whether the processing component should use supplied properties to refresh currently stored properties or not.

- *DELETE_ACTION* tells the processing component to remove this variable from its persistent storage system (typically the database). If the variable appears again, it will be recreated.

The raw values are sent through several communication channels. That is why they have a built-in serialization / unserialization process which converts them to or from strings. The syntax is very straight forward:

```
(+<flag>\t)?<timestamp>\t<groupname>\t<variable>\t<value>(\t<property>=<value>)*
```

The `\t` represents an ASCII Tab character (code #9).

The facilitative prefix `+<flag>` is the action code assigned to the raw value. The 2 known actions are:

- `+r` for *FORCE_REFRESH_ACTION*
- `+d` for *DELETE_ACTION*

The following are RawValue examples and their meaning:

```
+r 1199145600 group MyVariable1 100 device=myDevice name=MyVariable1
```

This RawValue's variable properties will be refreshed (`+r`). Its timestamp (1199145600) corresponds to January 1st, 2008, 00:00. The variable's retention group is *group*. The value is 100. Supplied properties are: *device* (myDevice) and *name* (MyVariable1).

```
+d 0 group MyVariable2 0
```

This RawValue's variable will be deleted from the Backend (`+d`). Its timestamp (0) corresponds to January 1st, 1970, 00:00. The variable's retention group is *group*. The value is 0 (a value *MUST* be supplied, even if the variable will be deleted from the Backend).

```
+r -1 group MyVariable3 0 device=myDevice3 name=MyVariable3 customer=Acme Engineering Inc.
```

This RawValue's variable properties will be refreshed (`+r`) only. The variable's retention group is *group* and its timestamp is negative, therefore the supplied value will be overlooked. However, a value *MUST* still be supplied. The refreshed properties are *device*, *name* and *customer*

```
1199145600 group MyVariable4 98 device=myDevice4 name=MyVariable4
```

This RawValue's variable properties will not be refreshed. Supplied properties are: *device* (myDevice4) and *name* (MyVariable4). The variable's retention group is *group* and its timestamp (1199145600) corresponds to January 1st, 2008, 00:00. Its value is 98.



Duration Notation

In some configuration files, durations can be entered in a specific format that greatly improves their readability.

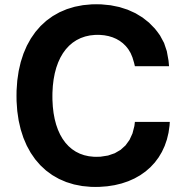
Instead of giving a number of seconds or milliseconds, the required input is a combination of number and units, those units being one of the following.

- `d` for days - ex: "1d"
- `h` for hours - ex: "6h"
- `m` for minutes - ex: "4m"
- `s` for seconds - ex: "300s"
- `ms` for milliseconds - ex: "100ms"; this last unit is not available when the field expects a number of seconds.

For instance, instead of writing "108000" which corresponds to a number of seconds equals to 30 hours, one can write either "30h", or "1d6h", or any other combination using these units.

Note: Such a notation is not available everywhere. Please refer to samples or corresponding documentation

to see if it applies.



Built-in Collecting subcomponents

1 Socket DCA

This collector listens for raw data on a *TCP/IP* socket. It works like the *Socket CI* but is designed for the collecting component. It is the preferred method to push data (standard raw value serialized format, see section [A](#) on page [37](#)) into *APG* Backend since you can use any previously designed collecting subcomponent of the collecting framework. This collector can also be used to create some intermediate collector to merge several collecting sources into a single stream.

To use this collector, simply declare it in the collecting configuration file as shown in the following example:

```
<collector enabled="true" name="..." next="..."  
  conf="Collector-Manager/Default/Socket-Collector/conf/socketcollector.xml" />
```

You have to specify the listening port in the collector configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE config SYSTEM "server.dtd">
<config>
  <listen>2003</listen>
</config>
```

Optionally, you can specify the character set used for value decoding (default is UTF-8).

```
...   <listen encoding="UTF-8">2003</listen>
...
```

2 File DCC

This connector prints any data it receives to a file or the console in a human readable form. It is useful to setup a collecting graph and check the collected data.

To use this connector, simply declare it in the collecting configuration file as shown in the following example:

```
<connector enabled="false" name="..."
  conf="Collector-Manager/Default/File-Connector/conf/file-connector.xml" />
```

To choose between error and standard output, simply write `err` or `out` in the `systemstream` element.

```
<?xml version="1.0" encoding="UTF-8"?>
<config
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.watch4net.com/APG/Connector/FileConnector_file-connector.xsd"
  xmlns="http://www.watch4net.com/APG/Connector/FileConnector">
  <output>test.log</output>
</config>
```

3 Out DCC

This connector prints any data it receives to the standard or error output in a human readable form. It is useful to setup a collecting graph and check the collected data.

To use this connector, simply declare it in the collecting configuration file as shown in the following example:

```
<connector enabled="false" name="..."
  conf="Collector-Manager/Default/Out-Connector/conf/outconnector.xml" />
```

To choose between error and standard output, simply write `err` or `out` in the `systemstream` element.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE config SYSTEM "outconnector.dtd">
<config>
  <systemstream>err</systemstream>
</config>
```

4 Socket DCC

This connector serializes raw values and writes them into a *TCP* socket. This is the default connector used on deployment case where collecting components and processing components are separated. This subcomponent should be used with the *Socket Collector Controle Interface*.

To use this connector, simply declare it in the collecting configuration file as shown in the following example:

```
<connector enabled="true" name="..."
  conf="Collector-Manager/Default/Socket-Connector/conf/cacheconnector.xml" />
```

The configuration file specifies on what host/port the connector will write data. For performance tuning on low latency links, you can set the buffer size to higher values.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE config SYSTEM "server.dtd">
<config>
  <host>host.com</host>
  <port>2000</port>
  <bufferSize>1024</bufferSize>
</config>
```

This connector sends raw values using the raw value serialization format. The encoding can be specified with the `encoding` element (default is UTF-8). Targeted component must be configured correctly in order to properly decode received values.

```
...  
  <encoding>ASCII</encoding>  
...
```

D

Built-in Processing Subcomponents

1 Socket CI

The socket control interface reads serialized raw values from a *TCP/IP* socket and submit them to the processing component.

To use this control interface, simply declare it in the processing component configuration file as shown in the following example:

```
<control-interface name="Socket-Collector" type="SocketCollectorInterface" config="conf/
socketinterface.xml" />
```

The configuration of this control interface is very simple. You simply have to provide the port the socket will be bound to.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE config SYSTEM "server.dtd">
<config>
```

```
<listen>2000</listen>
</config>
```

Optionally, you can specify the character set used for value decoding (default is UTF-8).

```
...
<listen encoding="UTF-8">2003</listen>
...
```

2 Crossed Failover Socket CI

The crossed failover socket control interface reads serialized raw values coming from many sources from a *TCP/IP* socket and submits to the processing component only data coming from the active source, while keeping data coming from non-active sources in case the active source becomes unavailable.

To use this control interface, simply declare it in the processing component configuration file as shown in the following example:

```
<control-interface name="Crossed-Failover-Socket-Collector" type="
CrossedFailoverSocketCollectorInterface" config="conf/crossedfailoversocketinterface.xml" />
```

The configuration of this control interface is very simple. You simply have to provide the port the socket will be bound to and the configuration file of the crossed failover.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE config SYSTEM "crossedfailoverserver.dtd">
<config>
  <listen>2002</listen>
  <crossed-failover-conf>conf/crossed-failover.xml</crossed-failover-conf>
</config>
```

Optionally, you can specify the character set used for value decoding (default is UTF-8).

```
...
<listen encoding="UTF-8">2003</listen>
...
```

For help configuring the `crossed-failover.xml` file, refer to `APG-Crossed-Failover-Filter.pdf`.

3 Telnet CI

The telnet control interface is designed to receive orders from a telnet session.

To use this control interface, simply declare it in the processing component configuration file as shown in the following example:

```
<control-interface name="Telnet-Collector" type="TelnetControlInterface" config="conf/telnetinterface.xml" />
```

The configuration of this control interface is very simple. You simply have to provide the port the socket will be bound to.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE config SYSTEM "server.dtd">
<config>
  <listen>2001</listen>
</config>
```

The telnet interface can receive several orders (case insensitive commands):

- COMMIT followed by a space separated list of groups order the cache to commit all data for each specified group.
- COMMITALL does the same but for all groups.
- CLEANALL asks the cache to order the storage connector to clean outdated data
- RESTART does a clean restart of the cache, re-reading the configuration (be careful that the cache will failed to restart if some configuration files get corrupted)
- STOP shutdowns the processing component in a clean manner (no data loss).
- STOPNOW shutdowns the processing component in the fastest way (some data loss may occur).
- BYE closes the connection

4 MySQL Storage Connector

The *MySQL Storage Connector* is the database manager specialized for *MySQL* database. It uses optimized procedures in order to get the best of a *MySQL* database.

To use this storage connector, simply declare it in the processing component configuration file as shown in the following example:

```
<control-interface name="MySQL" type="MySQLStorageConnector" config="conf/mysql.xml" />
```

The *MySQL Storage Connector* is configured by a simple XML file which contains database server credential and some specific configuration which should not be changed in most cases.

```
<?xml version="1.0" encoding="UTF-8"?>
<mysql xmlns="http://www.watch4net.com/APG/Backend" xmlns:xsi="http://www.w3.org/2001/XMLSchema
-instance" xsi:schemaLocation="http://www.watch4net.com/APG/Backend_mysql.xsd">

    <connection url="jdbc:mysql://localhost:53306/apg?autoReconnect=true" username="apg"
password="watch4net">
        <retry delay="10s" />
        <retry delay="30s" />
        <retry delay="1m" />
        <retry delay="2m" />
        <retry delay="5m" />
        <email-alert>
            <from>from@your.company.com</from>
            <to>dba1@your.company.com</to>
            <to>admin2@your.company.com</to>
            <message-on-failure subject="[APG]_Database_Connection_Failure">Database connection
lost! Please check the database status as well as the backend logs for more information.</
message-on-failure>
            <message-on-recovery subject="[APG]_Database_Connection_Established">Database
connection re-established. Everything is fine.</message-on-recovery>
        </email-alert>
    </connection>

    <data encoding="UTF-8">
        <tables prefix="cache" engine="MyISAM" />
    </data>

</mysql>
```

The url, username and password are used to connect to the database server. The exact syntax of that URL is: jdbc:mysql://<server>:<port>/<db-name>?autoReconnect=true

The retry elements are used in the exact given order when the connection to the database is lost for some reason. At the end of the last retry, an email will be sent to the given recipients using the information in the email-alert tag and the Backend will continue to retry with the last delay value. The

delay attributes use the [Duration Notation](#) defined in appendix B page 41.

The data element contains settings related to the way *MySql* stores data internally. The `table-prefix` attribute is for internal use only and should not be modified. It basically controls the table name generation: `table-prefix` is the string which will be prepended to processing component internal tables. You can also choose the *MySql* storage engine in `storage-engine`.

5 Oracle Storage Connector

The *Oracle Storage Connector* is the database manager specialized for *Oracle* database. It uses optimized procedures in order to get the best of an *Oracle* database.

To use this storage connector, simply declare it in the processing component configuration file as shown in the following example:

```
<persistent-storage name="Oracle" type="OracleStorageConnector" config="conf/oracle.xml" />
```

The *Oracle Storage Connector* is configured by a simple *XML* file which contains database server credential and some specific configuration which should not be changed in most cases.

```
<?xml version="1.0" encoding="UTF-8"?>
<oracle xmlns="http://www.watch4net.com/APG/Backend" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:schemaLocation="http://www.watch4net.com/APG/Backend_oracle.xsd">

  <connection url="jdbc:oracle:thin:@localhost:1521:APG" username="apg" password="watch4net">
    <retry delay="10s" />
    <retry delay="30s" />
    <retry delay="1m" />
    <retry delay="2m" />
    <retry delay="5m" />
    <email-alert>
      <from>from@your.company.com</from>
      <to>dba1@your.company.com</to>
      <to>admin2@your.company.com</to>
      <message-on-failure subject="[APG]_Database_Connection_Failure">Database connection
lost! Please check the database status as well as the backend logs for more information.</
message-on-failure>
      <message-on-recovery subject="[APG]_Database_Connection_Established">Database
connection re-established. Everything is fine.</message-on-recovery>
```

```
</email-alert>
</connection>

<data remote="true" encoding="UTF-8">
  <tables prefix="DATA" />
</data>

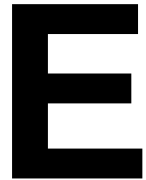
</oracle>
```

The `url`, `username` and `password` are used to connect to the database server. The exact syntax of that URL is: `jdbc:oracle:thin:@localhost:1521:APG`.

The `retry` elements are used in the exact given order when the connection to the database is lost for some reason. At the end of the last retry, an email will be sent to the given recipients using the information in the `email-alert` tag and the Backend will continue to retry with the last delay value. The delay attributes use the [Duration Notation](#) defined in [appendix B](#) page 41.

The `remote` attribute of the `data` element is used to tell the connector whether the Oracle database is on a remote server (`remote="true"`) or on the same server (no `remote` attribute or `remote="false"`). This changes the way the data is sent from the Backend to the database. Please keep in mind that the connector won't work if the database is on a remote server and this setting is not set to `true`.

The `encoding` attribute of the `data` element is used to tell the system equivalent encoding of the Oracle database. For example, if the Oracle database is using 'WE8ISO8859P1', the `encoding` should be set to 'ISO-8859-1'.



Backend webservice client

The Backend webservice client allows to call some commands also available through telnet. Moreover, it provides a way to merge two variable using their ID (variable field in the raw value).

The tool is located in the `bin` directory of the installed Backend instance.

Here are the available commands:

- `commitAll` order the cache to commit all data for all groups
- `commit (group name)` same as `commitAll` but only for specified groups. connector to clean out-dated data
- `cleanAll` does a clean restart of the cache, re-reading the configuration (be careful that the cache will failed to restart if some configuration files get corrupted)
- `clean (group name)` same as `cleanAll` but only for specified groups.
- `getVariables (variable name)` get informations about variables (numerical ID, last timestamp, group name)
- `deleteVariables (variable name)` delete variables

- `merge` (-t threshold -m old new) Merges variables. Old is the old variable name to merge with the new one. The threshold is a check for still collected data. The difference between the last timestamp of the two variables must be higher than the provided threshold to allow the merge operation.
- `nop` do nothing. Used to test communication with the webservice.

Note: using the tool with no parameters displays the help

Note: If the tool return errors, the Backend logs can normally provide more details about it.
