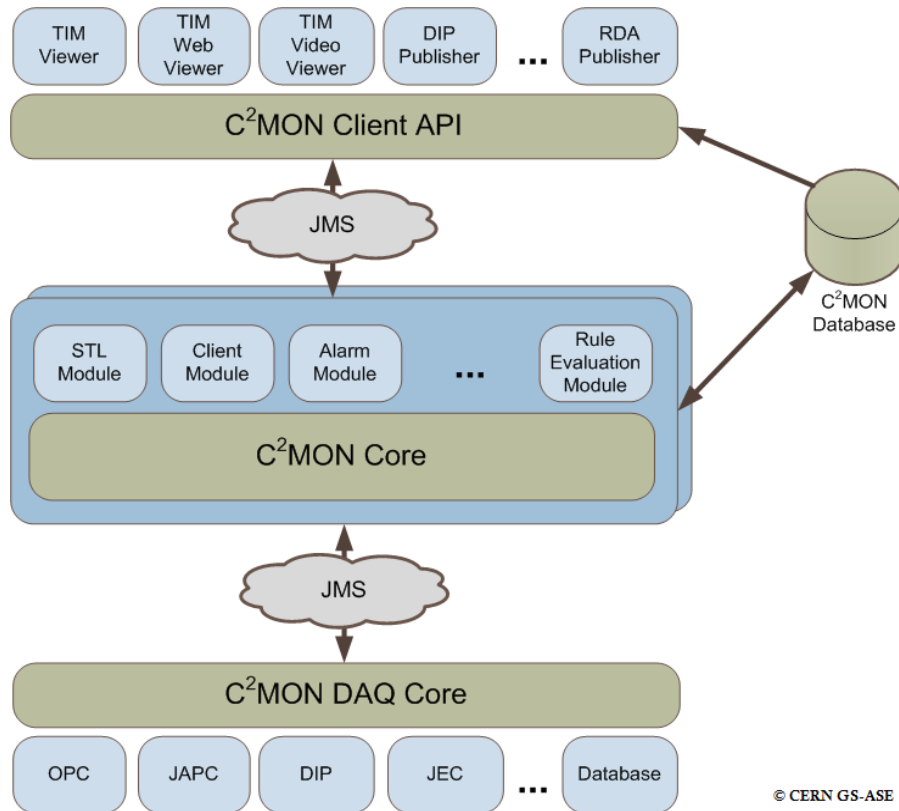


| | |
|--|----|
| 1. C2MON Home | 2 |
| 1.1 Data Acquisition | 2 |
| 1.1.1 C2MON DAQ core | 3 |
| 1.1.1.1 Data filtering on the C2MON data acquisition layer | 4 |
| 1.1.2 Creating a new DAQ module from scratch | 5 |
| 1.1.3 Emulator (Test) Module | 10 |
| 1.1.4 JEC Module | 11 |
| 1.1.5 JMX Module | 12 |
| 1.1.6 OPC Module | 14 |
| 1.1.7 Oracle Database Module | 16 |
| 1.1.8 SSH Module | 17 |
| 1.2 Platform Design Documentation | 19 |
| 1.2.1 Development tools | 19 |
| 1.2.1.1 Modules | 19 |
| 1.2.2 Server architecture | 20 |
| 1.2.2.1 Server threads | 20 |
| 1.2.2.2 Server locking strategy | 21 |
| 1.2.2.3 Server lifecycle management | 21 |
| 1.2.2.4 Writing a cache listener for a server cluster | 22 |
| 1.2.2.5 The server cache | 24 |
| 1.2.2.6 Server data configuration | 25 |
| 1.2.2.7 Server cache objects | 26 |
| 1.2.3 Server modules | 27 |
| 1.2.3.1 Cache module | 28 |
| 1.2.3.2 Cache DB access module | 28 |
| 1.2.3.3 Cache loading module | 28 |
| 1.2.3.4 Cache persistence module | 29 |
| 1.2.3.5 Configuration module | 29 |
| 1.2.3.6 Alarm module | 30 |
| 1.2.3.7 LASER server module | 30 |
| 1.2.3.8 Client module | 31 |
| 1.2.3.9 Java guidelines for C2MON server modules | 31 |
| 1.2.3.10 Supervision module | 32 |
| 1.2.3.11 Short-term-logging module | 32 |
| 1.2.3.12 Server common module | 32 |
| 1.2.3.13 Writing a new module | 32 |
| 1.2.3.14 Lifecycle module | 32 |
| 1.2.3.15 C2MON Bamboo dependency graph | 33 |
| 1.2.4 Server configuration | 34 |
| 1.2.4.1 List of server properties | 35 |
| 1.2.5 Platform management | 35 |
| 1.2.5.1 ActiveMQ broker management | 36 |
| 1.2.5.2 C2MON server management | 36 |
| 1.2.5.3 C2MON server recovery | 36 |
| 1.2.5.4 DAQ management | 36 |
| 1.2.6 Optimizing your C2MON setup | 36 |
| 1.2.6.1 Architecture scenario 1 | 37 |
| 1.2.6.2 Architecture scenario 2 | 38 |
| 1.2.6.3 Architecture scenario 3 | 39 |
| 1.3 Client Applications | 40 |
| 1.3.1 C2MON Client API | 40 |
| 1.3.1.1 Configuring the Client API | 40 |
| 1.3.2 C2MON Publishers | 40 |
| 1.3.2.1 DIP Publisher | 42 |
| 1.3.2.2 RDA Publisher | 43 |
| 1.3.2.3 LEMON Publisher | 45 |
| 1.3.3 C2MON Test Client | 45 |

C2MON Home

The C²MON project

The **CERN Control and Monitoring Platform C²MON** is a new solution proposed and adopted at the DIAMON/LASER TC presentation in March 2011, which is based on the idea of reusing common components of the Technical Infrastructure Monitoring (TIM) system for the new TIM2 and DIAMON2 services. C²MON covers all requirements, common to both services and allows creating highly available, reliable, scalable and flexible control solutions.



The C²MON platform uses a traditional 3-tier architecture, as displayed in the diagram above. The data acquisition layer provides solutions for acquiring data from a number of protocols/equipment: see [Data Acquisition](#) for details. The server architecture uses a Java Spring-based container and is designed to run in a clustered setup: see [Server architecture](#) for details. Internally, the server is broken down into a number of modules, including optional modules providing extra functionalities: see [Server modules](#) for documentation on each of these. Communication with the client layer is done via a provided C²MON Client API, which is documented here: [C2MON Client API](#).

Data Acquisition

Overview

C²MON is able to receive data from many different sources. This is achieved by so-called Data Acquisition (DAQ) modules. Each instance of a DAQ module is able to connect to many different devices at the same time. It is even possible to supervise these devices through alive timers. This assures that operators get directly informed, when a DAQ loses the connection to one of its sub-equipments.

Available Acquisition Modules

Below you find the documentation for all Data Acquisition (DAQ) modules available for C²MON:

- C2MON DAQ core
- Creating a new DAQ module from scratch
- DIAMON Module
- DIP Module
- Emulator (Test) Module
- ENS Module
- JAPC Module
- JEC Module
- JMX Module
- LASER DAQ Module
- OPC Module
- Oracle Database Module
- SSH Module

Documentation

- DAQ Core
- DAQ (re)configuration

Presentations:

- DAQ Core dynamic timedeadband filtering

Testing:

- DAQ Test case specification

C2MON DAQ core

To acquire data from a given source, the following steps need taking:

1. Implement a DAQ module for the given source, or use one of those provided.
2. Declare the DAQ Process in the C²MON server, together with all monitored points.
3. Start the DAQ process.

DAQ module are implemented on top of the C²MON DAQ core. The core implements the functionalities required for communicating with a C²MON server. This includes starting the acquisition process and receiving the DAQ configuration file from the server, sending incoming updates, receiving and passing on command requests etc.

The available source quality flags

The table below lists all quality flags that can be used within a DAQ module to describe the quality status of a given tag. The descriptions are taken from class [SourceDataQuality.java](#). Please have also a look to the next section which explains how the different quality codes are then interpreted by the C²MON server.

| Quality code | Quality flag | Description |
|--------------|------------------|--|
| 0 | OK | Quality code representing a VALID SourceDataTag value. |
| 1 | OUT_OF_BOUNDS | Quality code representing a SourceDataTag value that is outside the min/max range defined for the DataTag. |
| 2 | VALUE_CORRUPTED | Quality code representing a SourceDataTag value that has been corrupted before it was received by the DAQ. |
| 3 | CONVERSION_ERROR | This quality code must be set if source data cannot be converted to TIM data because source and TIM data type are not compatible (for example: float -> integer) |
| 4 | DATA_UNAVAILABLE | Quality code representing a SourceDataTag value that is currently not available from the source. Please use that error code with care, because it implicitly allows that it gets overwritten in the server cache by an update with an older time stamp. |
| 5 | UNKNOWN | Quality code representing a SourceDataTag value that is invalid for an unknown reason or for a reason not covered by the other quality codes. |
| 6 | UNSUPPORTED_TYPE | Quality code representing a SourceDataTag value that cannot be decoded because the data type sent by the source is not supported by the handler. |

| | | |
|---|--------------------------|--|
| 7 | INCORRECT_NATIVE_ADDRESS | Quality code representing a SourceDataTag value that cannot be acquired because of an error in the tag's hardware address. |
| 8 | FUTURE_SOURCE_TIMESTAMP | Quality code representing a SourceDataTag value that is acquired with the from a source with wrongly configured clock (in the future in relation to the DAQ host's time) |

Mapping of source quality flags on the C²MON server

Data acquisition layer and C²MON server/client layer have two separate set of quality flags. The reason is that DAQ can send quality information that are only interesting for the server. The C²MON server maps then the DAQ quality flags to the client quality as explained in the following table:

| SourceDataQuality (coming from DAQ) | TagQualityStatus (sent to client) |
|---|-----------------------------------|
| OUT_OF_BOUNDS | VALUE_OUT_OF_BOUNDS |
| DATA_UNAVAILABLE | INACCESSIBLE |
| VALUE_CORRUPTED CONVERSION_ERROR UNKNOWN UNSUPPORTED_TYPE INCORRECT_NATIVE_ADDRESS FUTURE_SOURCE_TIMESTAMP | UNKNOWN_REASON |



Info for programmers

The mapping logic is implemented by the [QualityConverterImpl](#) class which belongs to the [tim-server-cache](#) module.

Invalidation by the DAQ core

When the DAQ core is asked by the DAQ module to send a new value to the server, it will perform a number of quality checks first, invalidating tags when necessary. We list the invalidations occurring in the DAQ core:

1. **Future timestamp:** if the source timestamp provided by the DAQ module is more than **5 minutes in the future**, the tag is invalidated with quality FUTURE_SOURCE_TIMESTAMP. The old value is kept and a new invalidation source timestamp is set. Appropriate value description is set, containing the value of the invalidated tag as well as the incorrect timestamp. Example value description: *value: 130 received with source timestamp: [2012-11-26 13:46:01.995] in the future! No further updates will be processed and the tag's value will stay unchanged until this problem is fixed*
2. **Conversion error:** if the provided tag value cannot be converted by the DAQ core into the expected type, the tag is invalidated with quality CONVERSION_ERROR. The old value is kept, and a new invalidation timestamp is set (suggested change: keep old source timestamp).
3. **Out of bounds:** if the provided value is not within the predefined range - when provided - the tag will be invalidated with quality OUT_OF_BOUNDS. The old (last known in range) value is kept, and a new invalidation source time is set. The quality description string contains the value that has been filtered out plus the configured tag's min/max limits. Example value description: *source value: 130 is out of bounds (max: 100)! No further updates will be processed and the tag's value will stay unchanged, until this problem is fixed*

Filtering on the DAQ level

The DAQ core filters data before sending it the server, with the aim of only sending significant changes to the server. These mechanisms avoid flooding the server with unnecessary values and will often result in a significant reduction in the amount of data (our experience is of the order of 50 times less data after the filtering is configured properly). See [Data filtering on the C²MON data acquisition layer](#) for details.



Filtering is after invalidation!

DAQ core invalidation checks are made before any filtering checks are made. This means for example that an update with a future timestamp will result in an invalidation, even if the value is within a value deadband.

Data filtering on the C²MON data acquisition layer

A number of data filtering mechanisms are used within the C²MON DAQ core to avoid sending unwanted data to the C²MON server. This filtered data is then sent down a special channel, where administrators can monitor how much is being filtered out and for what reasons.

The filtering logic is applied in the DAQ core. In principal, all data sent from the data acquisition modules should go through this filtering (there is a method for bypassing the filtering, but it may be made private in the future!).



Alive and communication fault

In general alive and communication fault messages, sent using the provided methods from the DAQ module, have no filtering applied. However, the DAQ can be configured to filter out excessive alive sending by setting the **c2mon.daq.equipment.alive.filtering** system property to true. (in the DAQ start-up script you need to set `-Dc2mon.daq.equipment.alive.filtering=true`). If this property is set, the DAQ module will not send equipment alive messages more often than half of the time interval defined by the server (note: the DAQ knows the expected interval for each of its registered equipment unit, as it receives this information with each equipment unit configuration).

Example usecase:

The `c2mon.daq.equipment.alive.filtering` property is set. The expected equipment alive message (*heartbeat*) interval is: 60 000 ms. The DAQ receives from its equipment heartbeat every 15 000 ms. Due to the filtering policy enabled, the DAQ skips every second update. (*Heartbeat* is sent every 30 000 ms)

When sending a **valid** tag, 3 different criteria are applied, in the following order:

1. **Value deadband:** a value deadband is optionally configured on the Tag level. They are of 2 types, absolute and relative. If an *absolute deadband* d is set for a Tag, an incoming value will only be accepted if there is a difference of at least d between this new value and the previous one in the local cache. In the case of a *relative deadband* p (=a percentage), the difference is measured relative to the size of the previous value: for example, for a relative deadband of 20, a tag with current value 100 will only change if a value ≥ 120 or ≤ 80 is received.
2. **Repeated value:** an incoming value is rejected, if both the value and value description are the same as the previous value sent to the server.
3. **Time deadband:** if a time deadband t is set for a Tag, the value is recorded. After at most t milliseconds, this or the most recent value received will be sent. A time deadband never filters out invalid values, since we may want to record these (see below for the filtering of invalid values).



Static vs dynamic deadbands

A *static deadband* t is set on the Tag during server configuration and is permanently active for this Tag. Values will in general only be sent to the server every t milliseconds. As mentioned above, the filtering will not apply to invalid values.

A DAQ can be configured to apply *dynamic deadbands* to all its Tags. In this case, if too many values are received for a given Tag, a time deadband will be set on this Tag. This deadband will be removed if the rate descends below the specified rate again. The rate measured is the average number of updates sent to the server over a given period (notice values filtered for some other reason are not taken into account here). More details about dynamic deadbands can be found here:

<https://edms.cern.ch/document/1108486/1>

When sending an **invalid** tag, only a single filtering criteria is applied:

1. **Repeated invalid:** an invalid update is filtered out if the previous update sent to the server had the same quality flag and description (the quality flag is a code describing what is wrong with the value).

In order to trace the filtering taking place on the DAQ layer, all filtered values are sent down a separate JMS channel and logged in the database. Charts are then available for analysing what is being filtered and for what reasons.

Creating a new DAQ module from scratch

The following guideline explains you the main steps to create your own C2MON DAQ module. Traditionally a DAQ module was dealing with a certain type of equipment, for example PLC, OPC, etc. This is the reason why many of the classes and interfaces in the DAQ core contain the word "Equipment" in the name. But this does not mean that you cannot write a DAQ for a retrieving data from other sources like middle-ware protocols. An "Equipment" can in fact represent any kind of data source, like for instance: JMS brokers, DIP, JAPC, JMX, JSON messages, other in-house services, ...

Table of Content

- [Table of Content](#)
- [Creating the Maven POM configuration for your project](#)
- [Defining a new `HardwareAddress` class](#)
 - [Example for a `HardwareAddress` implementation](#)
 - [Best coding practice](#)
 - [Where to store the new `HardwareAddress` class?](#)
- [Extending the `EquipmentMessageHandler` class](#)
 - [How to implement `connectToDataSource\(\)` from `EquipmentMessageHandler`?](#)
 - [How to parse the configured list of data tags?](#)
- [The optional equipment handler interfaces](#)



Please notice, that this guideline does not cover how to package and deploy your project, neither how to start a DAQ module. If you are interested in these topics, please contact c2mon-support@cern.ch

Creating the Maven POM configuration for your project

Create a new Maven project and add the [tim2-daq-core](#) artifact as dependency. The c2mon-daq-core contains beside the abstract class [EquipmentMessageHandler](#) also the [DaqStartup](#) start-up class which contains the `main(String[])` method for launching later your DAQ module.

```
<dependency>
  <groupId>cern.tim2.tim2-daq</groupId>
  <artifactId>tim2-daq-core</artifactId>
</dependency>
```

Defining a new HardwareAddress class

Before starting with the implementation of a new DAQ module you should first think about the address format. Which information are required for subscribing to a value of your new data source? This is very important, since all configuration parameters will later be managed and provided by the C2MON server. The entire DAQ configuration is sent as XML message, but the DAQ core will de-serialize it back for you into simple Java object Beans. To do this, it will need to know the configuration class used for your DAQ module, which implements the [HardwareAddress](#) interface. Both, server and DAQ core have to have this class in their class-path to properly handle the serialization and de-serialization. For the de-serialization, the DAQ core makes use of [Java reflection](#). This is only possible, because the path to the concrete [HardwareAddress](#) class is provided within the [HardwareAddress](#) tag of the XML configuration (see example):

Example of a DataTag configuration for the JAPC protocol

```
<DataTags>
  <DataTag id="195429" name="UV.SPS.UACV1-00219_US$FARSUPSPS6:STATOBJ2" control="false">
    <data-type>Integer</data-type>
    <DataTagAddress>
      <HardwareAddress class="ch.cern.tim.shared.datatag.address.impl.JAPCHardwareAddressImpl">
        <protocol>rda</protocol>
        <service>rda</service>
        <device-name>CV.UACV1_219_AL2_CCC</device-name>
        <property-name>StsReg01</property-name>
        ...
      </HardwareAddress>
      ...
    </DataTagAddress>
  </DataTag>
  ...
</DataTags>
```

Example for a HardwareAddress implementation

Let's have a look at the existing class [DBHardwareAddressImpl](#) which is used by the [Oracle Database Module](#) (DB DAQ). To subscribe a DataTag the DB DAQ needs only one parameter:

| Parameter | Type | Description |
|------------|--------|---|
| dbItemName | String | Name of the DB field to which the given tag should subscribe to |

The corresponding [DBHardwareAddressImpl](#) class has thus only to define this single field.



The visibility scope of all parameters have to be set to protected

Please keep in mind, that the visibility scope of a parameter cannot be set to `private`, since this would disturb class introspection through reflection during the de-serialization phase.

DBHardwareAddressImpl Class

```
package ch.cern.tim.shared.datatag.address.impl;

import ch.cern.tim.shared.common.datatag.address.DBHardwareAddress;

public class DBHardwareAddressImpl extends HardwareAddressImpl implements DBHardwareAddress {

    /** Serial serial version UID */
    private static final long serialVersionUID = 1L;

    /** The name of the DB field to which the given tag should subscribe to */
    protected String dbItemName; // this field needs to be in scope protected because of Java
    reflection!

    /**
     * Default constructor
     */
    public DBHardwareAddressImpl() {
        // Does nothing
    }

    public DBHardwareAddressImpl(final String dbItemName) {
        this.setDBItemName(dbItemName);
    }

    @Override
    public String getDBItemName() {
        return dbItemName;
    }

    public void setDBItemName(final String dbItemName) {
        this.dbItemName = dbItemName;
    }
}
```

Best coding practice

The points listed below should help you defining your `HardwareAddress` class:

- As in the example above, your class should extend from [HardwareAddressImpl](#). Like this you don't have to deal with implementing the `HardwareAddress` interface and you can focus on defining your address format.
- Your class has to follow the [JavaBeans](#) standard in order to be compatible with the generic de-serialization procedure. This means for instance that you have to define getter- and setter-methods for all of your fields. Furthermore, you shall set the visibility of your fields to protected or public. Otherwise, the class introspection will not work.
- It is also good practice to define an interface for your class implementation. Later in your DAQ module you should then only use the interface and never access directly the class.

Where to store the new HardwareAddress class?

By convention we have one common place to put the concrete `HardwareAddress` implementations used by the services TIM and DIAMON, which is the following package in the `tim2-shared-common` library: [ch/cern/tim/shared/datatag/address/impl](#)

Extending the EquipmentMessageHandler class

The C²MON DAQ Core provides several interfaces that allows you defining the behaviour of your new acquisition module. Some interfaces are optional and don't need to be implemented straight away, like for instance the [ICommandRunner](#) interface for supporting commands. The only mandatory class which you have to extend is the [EquipmentMessageHandler](#). The abstract `EquipmentMessageHandler` class is the general superclass for all DAQs modules. It provides different methods to access parts of the core like for instance the configuration. The following four abstract methods have to be implemented within your DAQ module instance:

```

import cern.tim.driver.common.EquipmentMessageHandler;
import cern.tim.driver.tools.equipmentexceptions.EqIOException;

public class DemoMessageHandler extends EquipmentMessageHandler {

    /**
     * When this method is called, the EquipmentMessageHandler is expected
     * to connect to its data source. If the connection fails (potentially,
     * after several attempts), an EqIOException must be thrown.
     *
     * @throws EqIOException
     *         Error while connecting to the data source.
     */
    @Override
    public void connectToDataSource() throws EqIOException {
        // TODO Implement here the connection to your data source

        // TODO
    }

    /**
     * When this method is called, the EquipmentMessageHandler is expected
     * to disconnect from its data source. If the disconnection fails,
     * an EqIOException must be thrown.
     *
     * @throws EqIOException
     *         Error while disconnecting from the data source.
     */
    @Override
    public void disconnectFromDataSource() throws EqIOException {
        // TODO Handle data source disconnection
    }

    /**
     * This method should refresh all cache values with the values from the
     * data source and send them to the server.
     */
    @Override
    public void refreshAllDataTags() {
        // TODO Handle here the data refresh request. To communicate with the
        //       server you have to make use of the other methods provided by
        //       this class.
    }

    /**
     * This method should refresh the data tag cache value with the value
     * from the data source and send it to the server.
     *
     * @param dataTagId The id of the data tag to refresh.
     */
    @Override
    public void refreshDataTag(long tagId) {
        // TODO Handle here the data refresh request for the given tag id. To
        //       communicate with the server you have to make use of the other
        //       methods provided by this class.
    }
}

```




How does it work internally?

You have to be aware that your `EquipmentMessageHandler` class will be part of the DAQ configuration. So, only at runtime and **after** having received the XML configuration from the server, the DAQ Core will know where to find your handler implementation.

The reason behind this is that on single DAQ process can theoretically deal with many different types of `EquipmentMessageHandler` classes at the same time. Moreover, those can be changed for the given process simply by changing its configuration.

```

...
<EquipmentUnit id="1234" name="E_DEMO_DEMO1">
  <handler-class-name>cern.c2mon.driver.demo.DemoMessageHandler</handler-class-name>
  ...
</EquipmentUnit>
...

```

How to implement `connectToDataSource()` from `EquipmentMessageHandler`?

The `connectToDataSource()` method is one of the abstract methods provided by the `EquipmentMessageHandler`. The method is called when the core wants your module to **start up** and to **connect to the data source**. In fact the name is therefore maybe a bit misleading, because in practice this method is the main initialization point of your module. So, this method does normally the following steps:

1. retrieves the data source configuration from the DAQ Core by calling `getEquipmentConfiguration()`
2. handles the connection to the data source
3. parses the list of `ISourceDataTag` objects and creates for each of them a subscription to the data source
4. registers the other equipment handlers (if implemented), like for instance the `ICommandRunner` or the `IEquipmentConfigurationChanger`

This is of course a lot for a single method. Best practice is therefore to delegate the work to sub classes in order to simplify the main logic.



When is this method called?

The `connectToDataSource()` is only called once per equipment lifecycle. This means at the very beginning during the DAQ process start-up phase or after an equipment reconfiguration.

How to parse the configured list of data tags?

Theoretically every tag can be configured with a different `HardwareAddress` type (see also [Defining a new HardwareAddress class](#)), even if this is not really done in practice.

```

@Override
public void connectToDataSource() throws EqIOException {
    // TODO Implement here the connection to your data source

    HardwareAddress hardwareAddress;
    DemoHardwareAddress demoHardwareAddress;
    // parse HardwareAddress of each registered dataTag
    for (ISourceDataTag dataTag : getEquipmentConfiguration().getSourceDataTags().values()) {
        hardwareAddress = dataTag.getHardwareAddress();
        if (dataTag.getHardwareAddress() instanceof DemoHardwareAddress) {
            demoHardwareAddress = (DemoHardwareAddress) dataTag.getHardwareAddress();

            // TODO Handle the subscription of this tag
        }
        else {
            String errorMsg = "Unsupported HardwareAddress: " + dataTag.getHardwareAddress().getClass();
            throw new EqIOException(errorMsg);
        }
    }
}

```

The optional equipment handler interfaces

The table below describes the different optional interfaces that can be implemented by a DAQ module in order to provide more functionality. However, they are not required for a correct run of your DAQ module. Any reuests made by the server for a non-provided functionality will simply be rejected.

| Interface | Description | Registration class |
|--------------------------------|---|--|
| ICommandRunner | Provides one method <code>runCommand()</code> which is called, whenever the server is asking your DAQ module for executing a specific command | IEquipmentCommandHandler which can be retrieved by the EquipmentMessageHandler |
| ICommandTagChanger | Provides three methods to deal with changes in the command configuration (add/remove/update) | IEquipmentConfigurationHandler which can be retrieved by the EquipmentMessageHandler |
| IEquipmentConfigurationChanger | Provides one method <code>onUpdateEquipmentConfiguration()</code> which is called, when a configuration change for your DAQ module occurred. | IEquipmentConfigurationHandler which can be retrieved by the EquipmentMessageHandler |



Do not forget to register your implemented interface!

It is not enough to implement one of the interfaces. Of course the class must also be instantiated and registered. The registration can be done through the registration class which is provided by the EquipmentMessageHandler.

Emulator (Test) Module

| | |
|----------------------------------|---|
| Developed by | Mark Brightwell, GS-ASE-SSE |
| CERN contact | TIM Support |
| Status | In test phase |
| Source code | SVN Link |
| Handler class | ch.cern.tim.driver.testhandler.TestMessageHandler |
| HardwareAddress class | No Hardware class needed! |
| Supports dynamic reconfiguration | NO |

Overview

The TestMessageHandler simulates the behavior of a DAQ connected to some equipment. It reads a DAQ configuration XML file to determine the tags that the equipment should produce, and sends simulated values of these to the application server. A number of parameters added to the DAQ configuration file determine how the simulated values are generated. The simulator makes us of the declared min-max interval and value-deadband of a tag when choosing a value.

Configuration

The configuration requires two additions to the DAQ XML configuration file: first, the content of the `handler-class-name` tag must be replaced by the TestMessageHandler class name (ch.cern.tim.driver.testhandler.TestMessageHandler). Secondly, the required parameters must be added to the `address` field of the EquipmentUnit (these must be separated by semi-colons and be in the format parameter=value).

The Equipment Handler class

Set the handler-class-name to ch.cern.tim.driver.testhandler.TestMessageHandler

Equipment Hardware Address

All the parameters below are compulsory and must be included in the address field of the EquipmentUnit. The probability values are specified using a double between 0 and 1. The switchProb is specific for boolean tags.

| Parameter | Type | Mandatory? | Explanation | Example |
|-----------|--------------|------------|--|---------|
| interval | milliseconds | Yes | the time between the points at which the system generates events | 1000 |
| eventProb | double | Yes | the probability of the equipment sending a tag update to the DAQ | 0.15 |

| | | | | |
|-----------------|--------------------------|-----|--|-------|
| inRangeProb | double | Yes | if an update takes place, the probability of this update being in the min-max range of the tag | 0.9 |
| outDeadBandProb | double | Yes | if an update takes place within the min-max interval, the probability of the update being *outside* the value-deadband | 0.2 |
| switchProb | double | Yes | for boolean tags, the probability of the tag switching value | 0.01 |
| startIn | minutes in double format | Yes | at start-up, the time before the DAQ starts generating values | 1.5 |
| aliveInterval | milliseconds | Yes | the interval between <i>¿</i> equipment alive¿ messages to the server | 30000 |

Commands

The test handler does *not* support commands. Command tags specified in the DAQ configuration file will simply be ignored.

JEC Module

| | |
|---|--|
| Developed by | Andreas Lang (first version by Joao Simoes), GS-ASE |
| CERN contact | TIM Support |
| Status | Operational |
| Source code | SVN Link |
| Design documentation | EDMS Link |
| Handler class | ch.cern.tim.driver.jec.JECMessageHandler |
| HardwareAddress class | ch.cern.tim.shared.datatag.address.impl.PLCHardwareAddressImpl |
| Supports dynamic reconfiguration | NO |

Overview

The **Java Equipment Controller** (JEC) was developed by GS-ASE-SSE in order to connect Siemens S7 PLCs to TIM. JEC consists of a package to be installed in the PLC as well as a Java EquipmentMessageHandler for connecting to the PLC.

JEC PLC Library

| | |
|---|---------------------------|
| Developed by | Frederic Havart, GS-ASE |
| CERN contact | TIM Support |
| Status | Operational |
| Source code & installation guide | EDMS Link |

Configuration

Equipment Address

| Parameter | Default Value | RefDB Column |
|----------------------|--|---|
| plc_name | | MONEQUIP.IMPNAME1 |
| Protocol | SiemensISO or SiemensTCP or SchneiderTCP | MONEQUIP.EQPROTOCOL (codes from CG_REF_CODES.RV_DOMAIN (value=PLCPROTOCOL)) |
| Port | 102 or 6000 | MONEQUIP.PORTNO |
| Time_sync | Jec or ntp | MONEQUIP.TIMESYNC |
| Alive_handler_period | 5000 | MONEQUIP.ALIVEHANDPER |
| S_tsap | TCP-1 | MONEQUIP.SRCETSAP |

| | | |
|------------------|-------|----------------------|
| D_tsap | TCP-1 | MONEQUIP.DESTTSAP |
| Dp_slave_address | | <i>derived value</i> |

Tag Hardware Address

Please have a look into the [Design Documentation](#).

Documentation

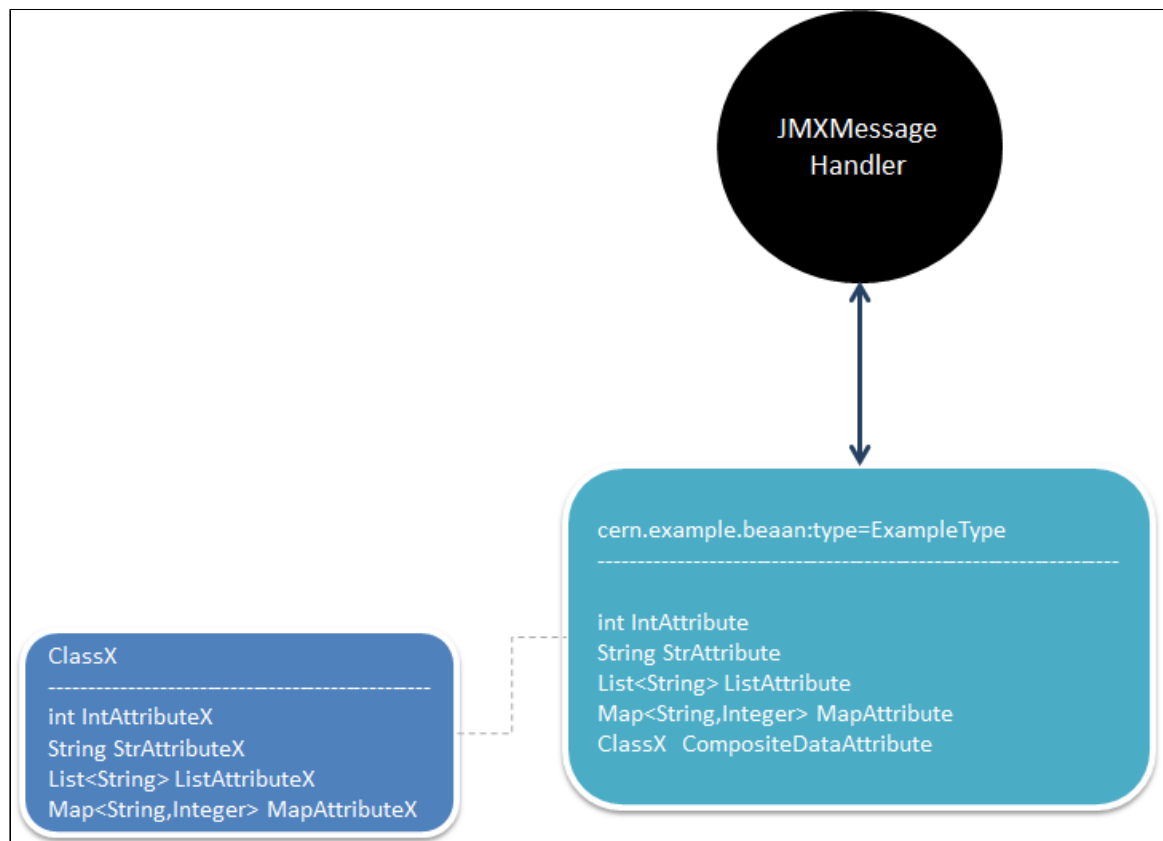
- [JEC Plc installation manual](#)
- [JEC DAQ redesign presentation](#)
- [JECMessageHandler data base constraints](#)

JMX Module

| | |
|---|--|
| Developed by | Wojtek Buczak |
| Status | Operational |
| Source code | SVN Link |
| Handler class | ch.cern.tim.driver.japc.JMXMessageHandler |
| HardwareAddress class | ch.cern.tim.shared.datatag.address.impl.JMXHardwareAddressImpl |
| Supports dynamic reconfiguration | YES |

Overview

This EquipmentMessageHandler allows C2MON-based systems to acquire metrics from java processes via JMX. The metrics can be monitored via periodic polling, or via subscribing to JMX bean attribute change notifications, if this method is supported by a JMX bean.



Configuration

Equipment Address

The JMXMessageHandler expects to receive equipment address in a form of string with semicolon used as a separator. The sequence of parameters is important:

| index | name | Mandatory? | Description |
|-------|----------------------|------------|--|
| 0 | service URL | YES | JMX service URL, e.g: service:jmx:rmi:///jndi/rmi://cs-ccr-inca3:61699/jmxrmi |
| 1 | user name | NO | if authentication is required, this user name is used for JMX connection, otherwise, this parameter can be skipped |
| 2 | user password | NO | if authentication is required, this password is used for JMX connection. otherwise, this parameter can be skipped |
| 3 | default polling time | YES | For metrics that need to be polled, this parameter defines the default polling time (in ms). This parameter shall be declared as the last one. |

Example equipment addresses:

```
service:jmx:rmi:///jndi/rmi://cs-ccr-inca3:61699/jmxrmi;user1;password1;10000
service:jmx:rmi:///jndi/rmi://cs-ccr-inca1:61699/jmxrmi;10000
```

Tag Hardware Address

to be completed

| Parameter | Type | Mandatory? | Explanation | Default value |
|---------------|--------|------------|--|---------------|
| objectName | String | Yes | the JMX object identifier, e.g: cern.example.mbeans:type=Cache | |
| attribute | String | No* | the name of the attribute of the JMX bean. If a metric is supposed to be monitored via notifications, this field is mandatory. | |
| callMethod | String | No* | the mbean method to be called in order to get a value of a metric. Only non-argument methods are currently supported, and this method is used only if a metric is monitored via polling. If set, the attribute parameter does not have to be set | |
| receiveMethod | String | No | specifies the method how the JMX handler shall be getting values of the metric. Two supported methods are: poll and notification | poll |

* Possible combinations of attribute and callMethod parameters:

| attribute | callMethod | description |
|-----------|------------|--|
| defined | null | the metric's value will be the value of an mbean attribute |
| null | defined | the metric's value will be determined by call to a mbean's method |
| defined | size | the mbean should expose given attribute, and the attribute should be a Collection. Metric's value will be the current size of the collection |
| defined | length | the mbean should expose a given attribute, and the attribute should be an array of primitives or objects. Metric's value will be the current size of the array |

example hardware addresses

- example 1

Periodic polling of a metric, provided by bean: *cern.accdm.logging.process:name=subscriberProcess* as a field of a map: *ParametersErrors* The map key to be used to get the value of a metric is: *LHC.STATISTICS.APP/Logging*

```
<HardwareAddress class="ch.cern.tim.shared.datatag.address.impl.JMXHardwareAddressImpl">
  <object-name>cern.accdm.logging.process:name=subscriberProcess</object-name>
  <attribute>ParametersErrors</attribute>
  <map-field>LHC.STATISTICS.APP/Logging</map-field>
</HardwareAddress>
```

- example 2

AttributeChangeNotification subscription of a metric, provided by bean: *cern.accdm.logging.process:name=subscriberProcess* as a field of a map: *ParametersErrors* The map key to be used to get the value of a metric is: *LHC.STATISTICS.APP/Logging*

```
<HardwareAddress class="ch.cern.tim.shared.datatag.address.impl.JMXHardwareAddressImpl">
  <object-name>cern.accdm.logging.process:name=subscriberProcess</object-name>
  <attribute>ParametersErrors</attribute>
  <map-field>LHC.STATISTICS.APP/Logging</map-field>
  <receive-method>notification</receive-method>
</HardwareAddress>
```

Useful Links

- [Overview of JMX technology, by Oracle](#)

OPC Module

| | |
|---|--|
| Developed by | Andreas Lang, CERN |
| CERN contact | TIM Support |
| Status | Operational |
| Source code | SVN Link |
| Design documentation | EDMS Link |
| Handler class | ch.cern.tim.driver.opc.OPCMessageHandler |
| HardwareAddress class | ch.cern.tim.shared.datatag.address.impl.OPCHardwareAddressImpl |
| Supports dynamic reconfiguration | NO |

Overview

This Module allows TIM to interface with a range of industrial SCADA systems via OPC, which has become an industry standard for Windows-based SCADA system. The OPC DAQ module was primarily developed for the Cooling and Ventilation (EN-CV) group in order to be able to acquire data from and send commands to their park of Wizcon stations. In general the DAQ can be used to access any kind of OPC conform system like PCvue32, PVSS or other SCADA systems supporting OPC DA 2.0 or higher.

Supported protocols

The OPC DAQ Module was rewritten in 2011 and supports now the following three different OPC protocols

| Protocol | Description |
|----------|---|
| DCOM | <p>The classic OPC protocol which is still the most used one. DCOM, which originally was called "Network OLE", extends Microsoft's COM, and provides the communication substrate under Microsoft's COM+ application server infrastructure. But using DCOM for OPC has three major disadvantages:</p> <ul style="list-style-type: none"> • DCOM has been deprecated in favor of Microsoft .NET Framework. • DCOM provides no security layer which means everybody can in principle retrieve and modify the OPC values • Difficult to set up and hard to debug |
| SOAP | <p>The XML (SOAP) based so called OPC XML Data Access (DA) specification was a first step of the OPC foundation to become more platform independent. The data structure is quite similar to the standard OPC DCOM DA client. The main differences are caused by the nature of SOAP.</p> |

| | |
|----------------------------------|---|
| OPC UA (Unified Architecture) | <p>The new standard in OPC communication which was introduced by the OPC Foundation. The modern and flexible design of this new protocol has many advantages:</p> <ul style="list-style-type: none"> • Replaces DCOM communication with SOAP (Web Services) and binary TCP/IP • Enables OPC in any operating system and language • Enables OPC in devices (embedded software) • Enables WAN (Secure Internet/Intranet/Extranet) connections • Improves Security Management • Combines all previous protocols to a common data model • Targeting to become an IEC standard <p>Read more about OPC UA >></p> |
|----------------------------------|---|

Configuration

Equipment Address

| Parameter | Optional? | Description | RefDB OPTID |
|--------------------|-----------------|---|-------------|
| URI | | URI of the OPC to connect to. The structure is <code>protocol://servername:port/path</code> . E.g. for dcom the URI could look like this: <code>dcom://VCVRMTBACNET01/SWToolbox.TOPServer.V5</code> . The other available protocols are http (uses the XML DA specification) and opt.tcp (uses the UA specification with TCP as base protocol). Optionally there can be a second URI specified to use in case the first OPC is not reachable. | ? |
| user | <i>Optional</i> | This can be used to specify the user for the authentication. It is optional to provide it since for example with OPC UA there can be other ways of authentication like a certificate. To provide Windows domain authentication the domain is added to the user string like this: <code>user@domain</code> . | ? |
| password | <i>Optional</i> | The password is usually provided if there is also user information provided. Again it might not be necessary for OPC UA. | ? |
| serverTimeout | | Inactivity timeout of the OPC server connection in milliseconds. If no data is received within this interval, the DAQ will check if the connection to the OPC server is still OK. <i>Recommended value:</i> 2000 to 5000 | 39 |
| serverRetryTimeout | | Connection retry timeout in milliseconds. If the connection to the OPC server is lost, the DAQ will wait <i>serverRetryTimeout</i> milliseconds between two consecutive connection attempts. <i>Recommended value:</i> 5000 to 30000 | 40 |

Tag Hardware Address

| Field | Data Type | Description |
|--------------------|-----------------------------|---|
| opcItemName | String | Full name of the publication within the OPC server. |
| addressType | enum(String, NUMERIC, GUID) | The type of the address. With OPC UA there are the additional types NUMERIC and GUID. STRING is also possible for UA and it is the only one used for DCOM and SOAP. |
| commandType | enum(Classic, Method) | In the former OPC specifications it was not defined how one has to call a procedure or a method. It was done by writing to special addresses in the OPC. OPC UA offers the ability to call methods which is why we have now an additional command type method. |
| namespace | int | This is only used for OPC UA else it is ignored. OPC UA provides the ability to have different namespaces with different values on the OPC server. For OPC UA this value is mandatory. |
| commandPulseLength | int | Only applies to boolean pulsed commands: delay in milliseconds between setting a command's value and resetting it to the inverse value (pulse). A pulse length of less than 500 ms is not recommended as it is quite likely that setting or resetting the value takes longer than 500 ms, depending on the OPC server. (Only used for CLASSIC commands) |

Configuring Remote Access for J-Integra on the OPC Server host

The TIM OPC DAQ uses the [J-Integra for COM](#) package provided by Intrinsic software for communicating with remote OPC servers.

Troubleshooting

- [Connecting through Windows firewall with DCOM](#)

Useful links

- [OPC and DCOM Configuration on Windows 2008 and Windows 7](#)
- [OPC Foundation Homepage](#)
- [Prosyst OPC UA](#)
- [J-Integra for COM](#)
- [OPC UA evaluation by EN-ICE](#)

Documentation

- [OPC DA 3.00 Specification](#)
- [Thesis about the new OPC data acquisition module](#)

Oracle Database Module

| | |
|----------------------------------|---|
| Developed by | Aleksandra Wardzinska, GS/ASE |
| CERN contact | TIM Support |
| Status | Operational |
| Source code | SVN Link |
| Design documentation | EDMS Link |
| Handler class | ch.cern.tim.driver.db.DBMessageHandler |
| HardwareAddress class | ch.cern.tim.shared.datatag.address.impl.DBHardwareAddressImpl |
| Supports dynamic reconfiguration | NO |

Overview

The Oracle database acquisition module (DB DAQ) allows to monitor selected information stored in the client databases.

The DAQ layer in C²MON system gathers values of data tags (points) that are monitored. For DB DAQ a data tag is a property of the database that the client wants to monitor. It could be for example the information about the status of the database, or amount of elements stored in a given table. Each property has to be declared first in the C²MON reference database. It will then be loaded to a dedicated table on timdbdaq account. It is this table that is the core part of the DB acquisition. Each row represents a data tag. Whenever an update is performed, the information is propagated to the DB DAQ java process.

The DB DAQ process doesn't monitor the client databases directly, but gets information from a C²MON DB account dedicated for this purpose. It is the client that updates the values of their data tags on this C²MON DB account. The DB module sends then the received values of data tags to the C²MON server.

Configuration

For more detailed information about how to push data from your database to C²MON please read the [Design Documentation](#).

Equipment Address

The DBMessageHandler requires the following parameters in its *equipment address*:

| Parameter | Mandatory? | Description |
|------------|------------|--|
| dbUrl | Yes | The JDBC connection URL of the C ² MON database which is used as gateway for the client databases |
| dbUsername | Yes | The DB user name |
| dbPassword | Yes | the DB password |

Example


```
<address>
  dbUrl=jdbc:oracle:thin:@<c2mon_database>;dbUsername=<username>;dbPassword=<password>
</address>
```

DataTag Hardware Address

Each DataTag for the DBMessageHandler is defined through the DBHardwareAddress interface which provides only one parameter.

| Parameter | Type | Mandatory? | Explanation |
|------------|--------|------------|---|
| dbItemName | String | Yes | Name of the DB field to which the given tag should subscribe to |

Commands

The DB module does *not* support commands.

SSH Module

Overview

TIM uses a Secure Shell (SSH) data acquisition module for executing SSH commands. This DAQ is mainly used internally, e.g. for restarting DAQ modules, OPC servers etc. via simple commands sent from the TIM Viewer

| | |
|---|--|
| Developed by | Wojtek Buczak (BE/CO) |
| CERN contact | TIM Support |
| Status | Operational |
| Source code | SVN Link |
| Design documentation | SSHMessageHandlerDesignDescription |
| Handler class | ch.cern.tim.driver.ssh.SSHMessageHandler |
| HardwareAddress class | ch.cern.tim.shared.datatag.address.impl.SSHHardwareAddressImpl |
| Supports dynamic reconfiguration | NO |

Configuration

Equipment Address

The SSHMessageHandler requires the following parameters in its *equipment address*:

| Parameter | Type | Mandatory? | Description | RefDB OPTID |
|------------------------|--------|------------|--|----------------|
| default_server_alias | String | Yes | specifies the default host for the handler's DTs/CTs execution. The default server alias's value should have the following format: machine[:port] Unless different port is specified, the 22 is taken by default. | 58 |
| default_user_name | String | No | the default user name for SSH authorisation | 59 |
| default_user_password | String | No | the default user's password | 60 |
| default_ssh_key | String | No | the location of the file containing RSA private key used for the handler's SSH authentication. If specified, the handler uses the private/public keys authentication mechanism by default. It is possible to use the \$HOME (or \${HOME}) environment variable inside the default_ssh_key. The handler validates its value locally, i.e. against the system the DAQ is started on. | 63 |
| default_key_passphrase | String | No | if the private key requires a passphrase, it should be set at this point. | 62 |

Examples

Some examples of the equipment addresses are presented below:

Ex1)

```
default_server_alias=TIM_DAQTEST01:22;default_user_name=timtest;
default_user_password=password;
```

Ex2)

```
default_server_alias=TIM_DAQTEST01:22;
default_ssh_key = $HOME/.ssh/authorized_key.priv
```

Ex3)

```
default_server_alias=TIM_DAQTEST01:22;
default_ssh_key=${HOME}/.ssh/authorized_key.priv;default_user_name=timtest;default_ssh_passphrase=passphr
```

Ex4)

```
default_server_alias=TIM_DAQTEST01:22;
default_ssh_key=${HOME}/.ssh/authorized_key.priv;default_ssh_passphrase=passphrase
```

Comments to the above examples:

Presence of the default_ssh_key in an information for the handler, that the SSH public/private key-pair authentication mechanism should be used instead of the one involving the user-name and password. Thus, although placing attributes: default_ssh_key and default_user_password in one address it is not formally incorrect (and will not cause any errors) it is redundant, as the handler will never use the second one. The handler may, however use the default_user_name for the SSH private/public keys authentication, if it is not specified inside the CT/DT hardware address block. In some cases, the fields of the SSHEMH equipment address can be overloaded by the corresponding attributes of the DT/CT hardware address block. This situation will happen for example, when one SSH DAQ process supervises DTs or CTs that need to execute their system-calls on different hosts.

Tag Hardware Address

| Parameter | Type | Mandatory? | Explanation |
|---------------|---------|------------|---|
| systemCall | String | Yes | specifies the SSH command (a binary program or a shell script) to execute on a remote machine after the handler logs into it successfully. |
| protocol | String | Yes | specifies the communication protocol between the handler and the HL. The protocol field must be set: xml or simple-io. |
| server-alias | String | No | like the default_server_alias from the equipment address, it specifies the remote machine on which the system-call should be executed. If specified, it overloads the default one. |
| userName | String | No | if specified it overloads its default equivalent from the generic equipment hardware address |
| userPassword | String | No | if specified it overloads its default equivalent from the generic equipment hardware address. |
| sshKey | String | No | if specified it overloads its default one from the equipment hardware address. |
| keyPassphrase | String | No | if specified it overloads its default one from the equipment hardware address. |
| callInterval | Integer | No | concerns only DTs. It specifies the time interval (in seconds) for periodic DT value recalculation. All DTs supervised by the SSHEMH are treated in the same way, i.e. their values are periodically reset by executing SSH system-calls, which should result with obtaining current values for the related tags. |
| callDelay | Integer | No | concerns only DTs. It specifies the time (in seconds), how long the first execution of a DT's system-call should be postponed, after the handler's initialisation. After the first execution is done, the handler continues to periodically execute system-call, according to call-interval scheduling parameter. |

The XML-based protocol

If a Data or CommandTag is configured to use XML protocol for communication with its dedicated hardware/system, the handler expects to receive an XML feedback on standard IO, when executing an SSH operation.

The format of the message is defined by the DTD, described below:

```
<!DOCTYPE execution [  
  <!ELEMENT execution-status (status-code, status-description)>  
  <!ELEMENT status-code (#PCDATA)>  
  <!ELEMENT status-description (#PCDATA)>  
  <!ELEMENT value (#PCDATA)>  
  <!ATTLIST value type (java.lang.Integer | java.lang.Boolean |  
    java.lang.Long | java.lang.Float | java.lang.Double |  
    java.lang.String) #REQUIRED>  
>
```

The element **execution-code** is mandatory and should contain an integer value of value 0 or -1. Value 0 indicates status: SUCCESS, while -1 indicates: FAILURE. The **status-description** element (not obligatory) carries additional information, such as e.g. explanation of the FAILURE execution status. If this concerns a DT, this information is used for tag invalidation, while with CTs this string is wrapped inside the command report. In both cases it is transmitted up to the client layer, so that the client can understand what happens in a clear way. The status description is ignored every time the invoked shell process exits with state: SUCCESS. The **value** element is obligatory only if using xml protocol with DTs. It wraps a tag value obtained by executing the tag's system-call. The value element should always have the type attribute, to inform the handler about the data type of the value. The possible types are defined by the DTD presented above.

Please refer to [SSHMessageHandlerDesignDescription](#) for details.

Platform Design Documentation

Content

- [Development tools](#)
- [Server architecture](#)
- [Server modules](#)
- [Server configuration](#)
- [Platform management](#)
- [Optimizing your C2MON setup](#)

Development tools

Some details on the development tools and how they are used...

Modules

Modules

C2MON modules Maven dependency graph

The C2MON modules are integrated with Maven, deployed to Nexus repository and automatically rebuild on build server (Bamboo) whenever some changes are committed. The build dependency of particular modules is presented on the following diagram:



C2MON modules description

- short description of each of the modules displayed on the diagram should be placed here...

Server architecture

C2MON server architecture

This page contains details of the general architecture of the C²MON server, including technical details on the threads and locking strategy.

- [Server cache objects](#)
- [Server data configuration](#)
- [Server lifecycle management](#)
- [Server locking strategy](#)
- [Server threads](#)
- [The server cache](#)
- [Writing a cache listener for a server cluster](#)

Server threads

This page gives an overview of the threads running in a C2MON server and the reason for these design choices.

JMS threads

These threads correspond to the MessageDrivenBean threads of a J2EE server. They are triggered by the arrival of a JMS message. They can be found in the C2MON server both on incoming messages from the DAQ and incoming client requests.

DAQ JMS threads (transacted)

On incoming DataTag updates, these threads will run at least until the DataTag has been updated in the cache. These JMS threads are transacted, meaning that if for some reason the cache update fails or is never performed, the JMS message will be re-delivered to a new consumer when one becomes available. Once the cache object has been updated, all registered listeners are informed. Listeners registered as *synchronous listeners* will be called on the JMS thread also (straight after the cache update). All other listeners will be notified *on their own thread(s)*, once they have been passed the newly updated cache object. For more details on cache listener design patterns, see [Writing a cache listener for a server cluster](#).

The above explanations also hold for incoming ControlTags, with the addition that after these tasks have been performed, the JMS thread will also run the required actions of the [Supervision module](#).

Client JMS threads (not transacted)

These threads are started similarly on incoming messages from the client layer. Mostly this happens in the Client module, but such threads are also started in the Video module for instance. These JMS threads are not transacted, and a failure will mean the Client request will fail (appears either as a timeout - if no answer is sent - or an error message, if the module is using the provided error message transmission).

Cache listener threads

As mentioned above, many cache listeners register as asynchronous listeners, or register for batches of updates (or batches of object keys). In all these cases, the listener is called on his own thread(s), and not on the original JMS thread. This has consequences in the case of a server crash, since these actions cannot be guaranteed to have been performed on all incoming data (see the ... section covering recovery after crash). The stopping and starting of these threads is the responsibility of the listener, which is passed a handle for this at registration time.

Supervision action threads

Miscellaneous threads

Most other threads in the server are performing action specific to a module, often for doing some regular check or clean-up operation. For example, the LASER module runs a separate thread to publish un-published alarms. These should be stopped correctly at shutdown by the module itself (or set as daemons for example).



When creating a new thread...

In general, when creating a new thread in some server module, try to give it a sensible name and document it in the module section of the wiki!

Server locking strategy

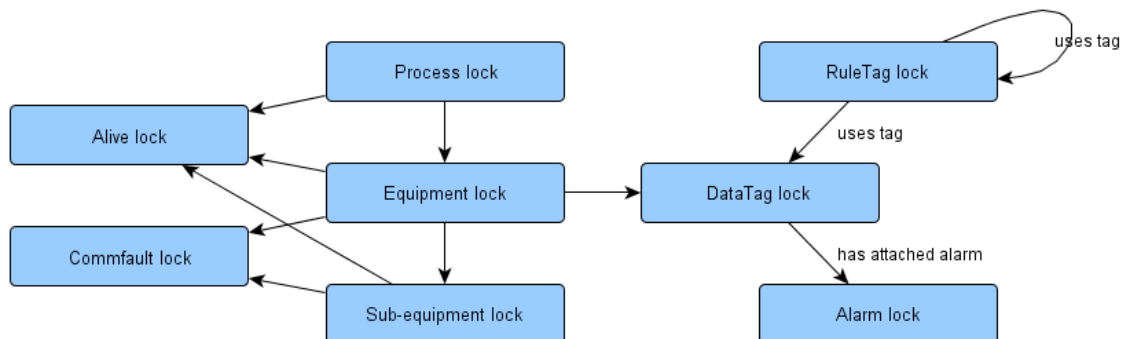
This page describes the java locking strategy used in the C2MON server. Locks are used both to synchronize a single server and synchronize across a cluster of servers. For locks to be effective across the cluster they must be in cluster-shared objects (use new Java Lock objects - for instance ReentrantReadWriteLock - since these are automatically picked up by Terracotta for cluster synchronization). To avoid deadlocks, the locks can only be called with the hierarchy detailed below (i.e. if one lock is taken after another, then it should be in this order). The hierarchy applies for both read and write locks.

Types of locks

- Each cache object (Cacheable.java) has an associated distributed lock
- Each cache has a distributed cache loading lock (in the DistributedParams object) used to synchronize cache loading a start-up across the cluster and to allow only a single object to be loaded from the DB (only happens at cache configuration - see [The server cache](#))

Lock hierarchy

The hierarchy has the cache loading locks at the top followed by the cache element locks. Within these, there is a strict lock hierarchy based on the cache element type: never take these locks in the incorrect order. The diagram below gives the lock hierarchy for the cache objects; the hierarchy for the cache loading locks is the same.



Beware of locks!

As opposed to a J2EE architecture, where cache object locking is managed for you, the C2MON developer is responsible for taking these locks in the correct order. If writing or modifying a complex part of code, where multiple cache objects are accessed, while at the same time some lock is held, make sure to first thoroughly test for deadlocks on your test system.

Server lifecycle management

The C2MON server uses the Spring lifecycle functionalities to manage the application lifecycle. From the lifecycle perspective, there are three kinds of components:

1. Spring beans that implement a Spring lifecycle interface (Lifecycle or SmartLifecycle)
2. Spring beans that do not implement a lifecycle interface
3. Objects that do not live in the Spring context

Objects that live in the Spring context

In general, all cache listeners and JMS components will be in the Spring context and should implement a lifecycle interface. This can only be avoided if the listener registers to the cache using a synchronous registration (since there is no thread to stop in this case). The lifecycle methods

should be used to

- stop/start JMS containers/DB connections at the right time
- stop/start threads used by the component (in particular, an asynchronous listener **must** call the lifecycle methods returned when registering to the cache!)

Best is to use SmartLifecycle, where the lifecycle phase can be specified (Lifecycle interface is the same as phase 0). These interfaces require the components to specify start/stop/isRunning methods.

For examples, see any existing modules: a good if complicated example is the LaserPublisherImpl (laser module); a simpler example is the AlarmPublisher (client module).

The following phases are used in the server

- phase -11: for components that must stop when all else is stopped, e.g. DB connections
- phase -10: for standard cache listeners
- phase -9: for cache listeners that use other components in phases -10 and -11
- phase 0: for cache listeners that write to the cache also (e.g. rule evaluation)
- phase 9:
- phase 10: for starting components gathering incoming data and requests (e.g. DAQ incoming JMS listener containers)
- phase 11: for final start-up components, such as server heartbeat

When starting, phases go from -11 to 11. When stopping, from 11 to -11.

In general, cache listeners will be in -9,-10,-11 (such as publishers) and incoming data components will be in 9,10,11. Phase 0 is for components that listen to the cache and modify other cache elements (e.g. rule evaluation)



Notice that the cache Spring beans (such as DataTagCacheImpl, CacheServiceRegistrationImpl etc) do *not* implement a lifecycle interface: this is important as the listeners usually depend on these, and the shutdown phase of the listener components would then be influenced by the shutdown phase of the cache beans: see the Spring documentation on bean lifecycle for details:

<http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/context/SmartLifecycle.html>

Spring beans with no lifecycle interface

If the bean has no threads to manage or other external resources, then there is probably no need to implement any lifecycle interface: recall, any initialization can be placed in an *init method* (annotated with `@PostConstruct`), and this code will be run before the bean is made available to others (so during context creation, as opposed to the SmartLifecycle methods).

As already mentioned, a good example of this is the C2MON cache beans (e.g. DataTagCacheImpl).

Objects outside the Spring context

A number of objects in the C2MON server do not live in the Spring context. These are often services that are created/destroyed during runtime of the application, for which the Spring container management makes less sense. The lifecycle of these objects will often be managed by some Spring bean. For these objects, we provide a C2MON lifecycle interface based on the Spring Lifecycle interface (we do not use the latter so that lifecycle interfaces indicate whether Spring is managing the lifecycle or not, and so as not to tie the code to Spring unnecessarily).

For example, this Lifecycle interface (`cern.tim.server.common.component.Lifecycle`) is used when registering an asynchronous cache listener: a Lifecycle interface is returned from the registration method. This handle should be used by the listener to stop/start the notification threads it relies on. See the CacheRegistrationService for examples of these registration methods, and the LaserPublisherImpl for how they should be used.

Writing a cache listener for a server cluster

This page describes what to keep in mind when writing a cache listener for a distributed C2MON cluster. A number of design choices exist, of which the following are the most important:

1. A synchronous listener
2. An asynchronous listener that does not access the cache
3. An asynchronous listener that access the latest cache object

In addition, options 2 & 3 can choose to register to receive collections of cache objects on a regular basis, rather than single cache objects.

We briefly describe the 3 options above.



Check locking!

When writing a cache listener, care must be taken - as always - to take locks in the correct order. In particular, a synchronous cache listener is called within a write lock on the updated cache object, so any locks acquired within this thread must be below in the lock hierarchy. If cache elements higher up the lock hierarchy need accessing (read or write!), the listener **must** register as an asynchronous listener (or access the objects in its own thread). See the following page for a description of the lock hierarchy: [Server locking strategy](#)

Synchronous listener

A synchronous listener is called *on the original JMS threads*. Incoming data is picked up by this thread, which puts the data into the cache, and calls the listener. A few comments:

- the listener call on an incoming DataTag is guaranteed to terminate successfully, since the JMS message is only acknowledge once the listener call returns, and will be redelivered on failure (so may be called twice with the same value if another synchronous listener fails for instance)
- a slow listener will gradually use up all the server threads, so can crash the server (warnings are currently sent by email if this occurs)
- a lock is held on the Tag during the call, and the Tag cannot be accessed by any other thread
- in particular, the order of the calls is the same as that of the data put in the cache, also *across a server cluster*

Notice that if listening to RuleTag updates also, the notification is then done on a separate rule-evaluation thread, so is not guaranteed to take place during a server crash (a server recovery action exists to correct this on restart, which can be called using a script option or a JMX call).

Example

An example of such a listener can be found in the available C2MON Alarm module: the AlarmAggregator implementation. This bean listens to changes in the Tag caches (ControlTag, DataTag and RuleTag) and evaluates any associated alarms. The alarm evaluation takes place in the original cache-update thread (the JMS thread in the case of a DataTag): once the call returns, the alarm is guaranteed to have been evaluated and put in the Alarm cache.



Hint

In general, use such a listener for internal server use, not for publication or interaction with other systems. The calls should return rapidly and not have any reason to block. This is also the easiest way of guaranteeing the order of these calls *across a server cluster*.

Asynchronous with no cache access

An asynchronous listener is called on *its own thread(s)*. When registering an asynchronous listener, you can specify the max number of threads you would like this listener to be called on.

- the listener is passed a *copy* of the cache object that has been updated
- the listener then performs its task on this object directly (it is not re-retrieved from the cache)



Hint

Use a single notification thread if the order is important for your listener: the updates will then arrive in the order they were put in the cache *locally*. Be careful here, since in a server cluster there is no order guarantee across the servers in this case: see the 3rd option below as a possible solution to this.

Example

A good example is publication modules, where the system on the opposite end can *filter out older updates*. This requirement is required for this module to work in a server cluster, since there is no guarantee objects will not overtake each other. The objects *server-timestamp* should be used for ordering the updates on the system that is reading the publications, since this timestamp corresponds to the order elements were put in the C2MON cache, which should always be considered correct.

Asynchronous listeners that accesses the cache

This listener design differs from the previous one in that it re-accesses the cache and retrieves a new copy of the cache object (in practice, only the id of the passed cache object is used). This is done so that the cache object is now *live in the cache, and the can be locked using its internal lock*.

This strategy guarantees the listener has:

- the most recent status of the cache object
- the cache object will not be updated until the listener has finished processing the update
- the locking takes place *across the C2MON server cluster*

**Hint**

This design choice should be used when the listener requires absolute synchronization across the server cluster and is best run on its own thread(s) (otherwise the synchronous option should be considered). For instance, a publisher that must order all publications across the server cluster may wish to follow this pattern: **but remember** this is only required if the publication client allow no possibility to communicate an ordering to it, typically by setting some timestamp using the publication API. If this possibility exists, use the C2MON server timestamp for this.

Example

A good example of this is the C2MON LASER publication module. The above pattern guarantees alarms are sent in a strictly synchronous manner across a cluster of servers (notice by "synchronous" here we mean *per* alarm). In addition to this, the LASER publisher and backup-publisher synchronize on their own shared lock across the server, to freeze all alarm sending during a backup generation and publication.

The server cache

Wrapper of Ehcache

The server cache implementation is a wrapper of the Java Ehcache library, which allows clustering across Java applications using the Terracotta technology.

The cache is loaded a server start-up from the DB. It is then persisted asynchronously to the database on a regular basis (10s): when correctly shutdown, the current status will be persisted; if a crash occurs (or kill is necessary), the server may not manage to save all the latest values to the DB.

When running a distributed cache, the cache is only loaded on the initial server start-up, since the distributed cache is persisted to disk and survives between restarts. Only if the cache is manually cleared after a shutdown will it be re-loaded at the next start-up.

The design does **not** allow for eviction from the cache. Eviction is not allowed since a lazy-loading design is then required and this places more constraints on the cache persistence to the DB. Instead, make sure the cache size is set large enough. If the memory is not sufficient to hold the cache, allow overflow to disk in the single server configuration, or run as a distributed cache (which is configured for disk persistence in our setup).

Cache API

The cache API can be found in the `cern.tim.server.cache` package in the `tim2-server-cache` project. This package contains all the interfaces to the caches themselves and the other services available for querying them. All caches are Spring beans and implement the `TimCache` interface, as well as a more specific interface for the given cache (e.g. `DataTagCache` for the cache containing the `DataTags`). This specific interface can then be used to inject the Spring bean into any other module. In addition, a facade bean is provided with extra methods for interacting with these cache element (e.g. the `DataTagFacade` bean).

The Tags in the system are spread across 3 different caches, depending on their type: the `DataTagCache`, the `RuleTagCache` and the `ControlTagCache`. Tag ids are unique *across* all these 3 caches (this is enforced in the database). The `TagLocationService` can be used to query all Tag caches given a Tag id, while the `TagFacadeGateway` provides a common gateway for querying/applying changes to Tags without necessarily being aware of their type.

Cache configuration

The DB is still used to guarantee the consistency of the *configuration* of cache elements: if an exception occurs during a cache reconfiguration, the affected cache elements are removed from the cache and reloaded from the DB. For this reason, the `TimCache` does provide a method for loading a single element from the DB into the cache. In this case, updates to the cache element may be overwritten if they have not been persisted to the DB in the meantime (DAQ restart will fix this).

Cache object loading vs configuration

As already mentioned, cache objects can be introduced into the system in 2 different ways:

1. at server startup, they can be loaded into the cache from the DB
2. at server runtime, they can be loaded into the cache by running a configuration

In both cases, cache objects are being created and put into the cache. However, it is important to notice these mechanisms share very little logic, must be kept consistent, and of course tested individually. Only the lowest level of object creation is shared: the constructors of the cache object implementation themselves. After this, the objects are handled differently:

1. for cache loading, the [Cache loading module](#) handles some initial post-DB-loading logic before passing the objects to the cache module itself, which puts them in the cache. Caches which require some complex global actions immediately after loading can do so using a custom bean that runs after cache creation: see for example the `RuleTagPostLoaderProcessor`, which sets all rule parent process ids at startup.
2. for cache configuration, the [Configuration module](#) manages the process. It calls the `createCacheObject` in the corresponding facade bean in the [Cache module](#).



Consistent cache object creation

Keep in mind that cache objects can be created at cache loading or during reconfiguration at runtime: these 2 mechanisms must be kept consistent and tested separately!

Server data configuration

The page describes the server configuration mechanism, that is how DAQ, Equipments, Datatags etc are added to the server at runtime. Further details can also be found in the documentation about the configuration module: see [Configuration module](#).

Server reconfigurations: general description

Configurations are specified in 3 tables in the database and loaded by passing the id of the configuration to the *Configuration Loader* application.

A configuration (specified by its id and listed in the TIMCONFIG table) is made up of an ordered collection of *Configuration Elements*. These are ordered by their ids (called *seqid* in the TIMCONFIGELT table). Details of each Configuration Element can be found in the 3rd final table TIMCONFIGVAL.

A Configuration Element is one of 3 types:

1. A CREATE element, which corresponds to the creation of a new cache element.
2. An UPDATE element, which corresponds to an update specification of an existing cache element.
3. A REMOVE element, which corresponds to the removal of an existing cache element.

Each type of element can then be applied to any of the following cache elements: Process, Equipment, SubEquipment, DataTag, ControlTag, RuleTag, Alarm.

Notice that REMOVE elements are applied recursively to all sub-elements: for example, the removal of a DataTag will result in the removal of the DataTag and all dependent Rules and Alarms; also, the removal of an Equipment results in the removal of all attached Sub-equipments, DataTags, etc.



Configuration design tip

When specifying a configuration in the database, think of designing it so that it can be rerun multiple times (for example, always include a "remove process" action before creating a new one). This can help recover from configuration errors and they can be re-used at a later date.

Transactions

In most cases, configurations are loaded in a transactional manner *on the Configuration Element level*, so an element is either applied in its entirety or left in its original state: this is true for all CREATE and UPDATE configuration elements. However, for REMOVE elements, recursive changes are committed *at each removal*: for instance, if a Tag removal results in an Alarm removal, a successful alarm removal will be committed even if the Tag removal fails (see details below). Similarly for an Equipment removal: all successful DataTag, Rule and Alarm removals that result from this will be committed, even if the final Equipment removal fails for some reason. This removal transaction policy means a REMOVE element may only be partially applied: however, applying once again the same remove element will in nearly all cases still access and attempt to remove all the leftover elements. **The only exception** to this is for control tags: these removals are committed *after* the Equipment/Process remove has been committed, so in the worst case you may end up with some "unreferenced" control tags still in the cache/DB. In this case, the Control tags would need removing using individual configuration elements (however, the only reason such an Equipment/Process removal should fail is (1) the DB becomes unavailable at that time or (2) these control tags are erroneously reference by some other Equipment/Process).

If a failure occurs during a transacted element, the DB transaction is rolled back and the server attempts to restore the cache to its original state.

Notice that **any failure** during a remove operation will **interrupt the removal**. For instance, if a DataTag removal fails during an Equipment removal, the remove operation will be interrupted immediately, leaving the Equipment in place with all remaining attached DataTags (and dependent rules and alarms). If a given object (e.g. DataTag) cannot be found this is not considered a failure: a warning is logged; if running in a larger removal, the latter will continue.

Tag removals

A Tag REMOVE will always result in the following actions, in this order (for DataTags, RuleTags and ControlTags)

This behaviour is the same whether the removal is a single REMOVE element or triggered by an Equipment removal.

1. remove all Rules (RuleTags) dependent on this Tag
2. remove all Alarms dependent on this Tag

This first call is called recursively on Rules dependent on Rules, removing any alarms dependent on these. Also, each individual removal is committed individually, whether the entire Tag removal is successful or not. For example, when removing a Tag with a single dependent Alarm, the Alarm is removed first and committed if successful, even if the actual removal of the Tag fails for some other reason.

If an error occurs during a Tag removal, the operation *is interrupted immediately*: that is, what is already removed is definitively committed, while any remaining Rules or Alarms that were not yet removed will remain attached to the Tag. However, the initial Tag is only removed *once all dependent Rules and Alarms have been removed successfully*. This guarantees no "lost" elements are left, since they can all be accessed from this Tag.

Finally, all Equipment keep a reference to the DataTags they contain. This reference is removed after the DataTag removal from the DB and cache has returned successfully. This means that if the thread crashes at that point, the Equipment will contain a reference to an unknown Tag: to fix this, the Equipment will need reconfiguring (remove and create). Alternatively, remove & reload the Equipment in the cache, manually from the JConsole.

Subequipment removals

A Sub-equipment consists essentially of a monitored alive tag and associated state tag, possibly with an attached commfault tag. A removal of this Sub-equipment will result in a removal of all these attached objects. The control tags are removed last, and in case of failure may remain unattached to any Sub-equipment. A Sub-equipment has no attached DataTags.

Equipment removal

As well as performing the removals described for Sub-equipments, an Equipment removal will first attempt to remove all attached DataTags. These DataTag removals recursively trigger Rule and Alarm removals: each individual removal is committed individually. Any failure immediately interrupts the overall removal (except if a given entity is not found, in which case a warning is logged but the removal continues).

Process removal

A process removal removes all attached Equipment before removing the Process itself. The Process control tags are removed last (state and alive tags). Internally the JMS subscription to the queue of incoming tags is closed.

Server cache objects

When adopting C2MON as the basis for a monitoring service, the only real constraint arises from data structures. Indeed, if the user were to make major changes to the data structure, much of the server core would need adapting to some extent. In this section we describe these data structures and their role in the C2MON architecture.

The Tag interface

The Tag interface embodies the common structure of 3 data type in the C2MON architecture: the DataTag, the ControlTag and the Rule(Tag). Tags should be thought of as follows: objects implementing this interface are destined to be updated, logged, published to clients and used in data views. They can also have associated Alarms attached. They will be able to use all the provided infrastructure of the C2MON architecture, including all the functionalities of the Client API.

The DataTag

The DataTag is a Tag coming from an external data source. It corresponds to a single external data point, represented by a Java primitive type (String, Float, Integer, etc.) It is attached to an Equipment, which embodies the type of data source it originates from. A DataTag has an *Address*, which contains the information needed for subscribing to this data point.



DataTag value type

Many functionalities in the C2MON server are only available for data points with *primitive* type values, such as Floats or Strings. However, the core C2MON could handle more complex data type without too many changes. DataTag values are stored as simple Java Objects.

The RuleTag

The RuleTag is a Tag internal to C2MON built on top of other Tags. Its value is specified by a Rule expression given in a specific C2MON rule grammar. This expression specifies how the Rule value is derived from the value of the Tags that feed into it. Changes to these Tags will result in an update of the Rule, based on the Rule expression.

The ControlTag

The ControlTag is any Tag used by the C2MON system for monitoring/publishing *internal states of the system itself* (we include here Equipment as part of the system, even if it may be shared with others). In other words, ControlTags are used by C2MON for *self-monitoring of some component of the system*. Making use of the functionalities available to all Tags, these values can then be logged and sent to C2MON clients for self-monitoring purposes. Alarms may also be attached to detect critical problems.

Equipment Alive Tags are a good example of a ControlTag, since they are used to monitor the status of the supervised Equipment. In a similar way, an Equipment Status Tag is used to publish the current status (running or down) of the Equipment.

The Equipment

The Equipment is the internal representation of some external data source. It is the highest level such representation, with the SubEquipment

coming below. Equipments always have an attached Status ControlTag and CommFaultTag. The status Tag contains the current status of the device, while the CommFaultTag can be used by the Equipment to notify a change in status. An Equipment may or may not have an attached Alive tag, although it is recommended to implement such a functionality in the EquipmentMessageHandler if possible.

The SubEquipment

The SubEquipment is attached to a given Equipment and represents a sub-system of this Equipment, whose status also needs monitoring (by the C2MON alive mechanism, as in the case of the Equipment). The same ControlTag types attached to Equipments can be attached to a SubEquipment. DataTags are not attached to a SubEquipment.



Future plans

As mentioned above, SubEquipment currently do not have any attached DataTags, which are only attached to an Equipment. This may be changed to allow invalidating of DataTags on SubEquipment failure.

The Process

The Process cache object is the internal C2MON representation of a Data Acquisition Process (DAQ). As such, it is an internal representation *of part of the system*, and is used exclusively for internal monitoring and management operations.

Server modules

C2MON Server modules

We provide some details of the various C2MON server modules. By C2MON server core, or core modules, we mean all modules that are required in any C2MON instance. All modules correspond to a single Maven artifact (and Java project). These can be found in SVN at svn.cern.ch/repos/tim2. The Java package structure is harmonized across all projects, so please follow the following guidelines when adding a new module: [Java guidelines for C2MON server modules](#).

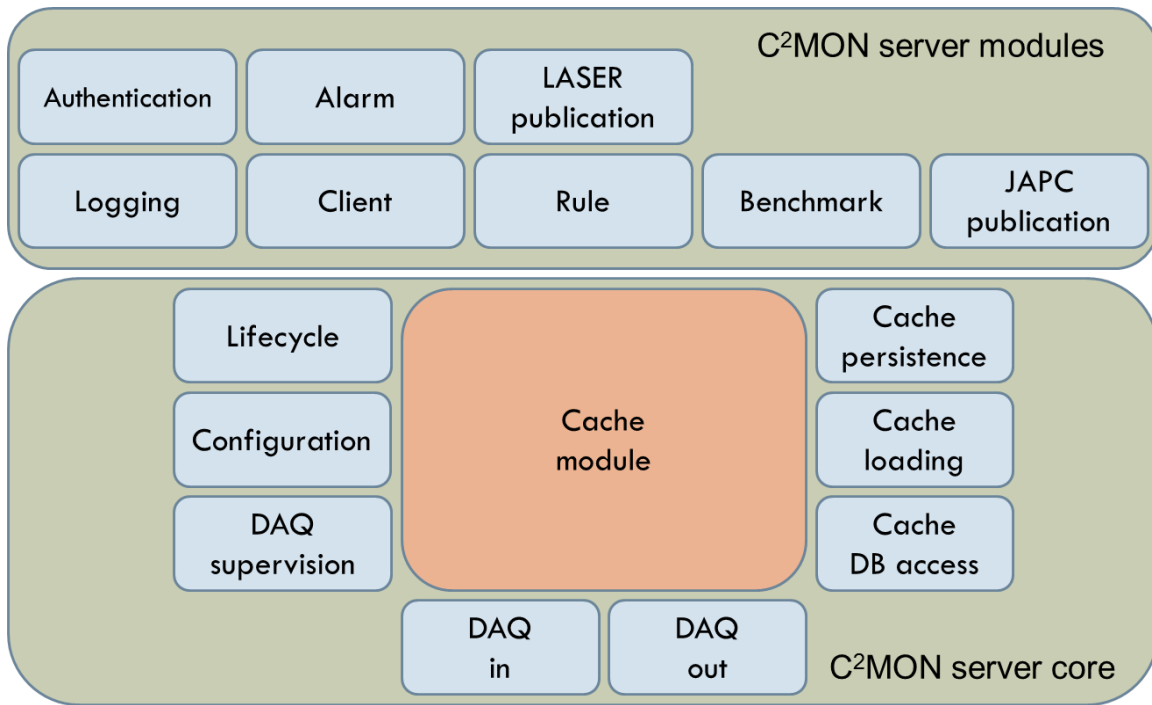
Core modules

- [Cache module](#)
- [Cache DB access module](#)
- [Cache loading module](#)
- [Cache persistence module](#)
- [Configuration module](#)
- [Server common module](#)
- [Lifecycle module](#)

Optional modules

- [Alarm module](#)
- [LASER server module](#)
- [Client module](#)
- [Short-term-logging module](#)

The diagram below gives an overview of the internal modular structure of the C2MON server.



Each module corresponds to a separate Maven artifact. An overview of the dependencies between these can be found here: [C2MON Bamboo dependency graph](#).

Cache module

Cache DB access module

The Cache DB access module manages the writing/reading of cache objects to the database (for details about these objects, see [Server cache objects](#)). These functionalities are exposed through "Mapper" beans that can be accessed through the interfaces in the `cern.tim.server.cache.dbaccess` package. All the implementation of the DB logic is contained in this module.

The implementation is provided and tested for an Oracle database (currently 11g).



Oracle independence

In principle, the database features used in this module are available on most databases. Transactions are not required. In practice, some of the statements may still contain Oracle-specific statements and need standardizing to run on other databases.

This module uses the Mybatis framework for accessing the database. The Mybatis mapper XML specifications are contained in the `sqlmaps` subpackage. Type handlers for handling the persistence of complex types can be found in the `type` subpackage.

Tests

The functionalities provided by this module are generally easy to test. The current tests run against the test Oracle database (may wish to move this to in memory DB once statements are standardized, or possibly run both tests).

Cache loading module

This module provides DAO beans for retrieving cache objects from the database. These DAO functionalities are then used by the cache module when loading the cache at server start-up. Caches containing few elements are loaded in 2 steps: (1) get all elements from DB - this is done by the Cache Loading module and (2) load them into the cache - this is done by the cache module itself. Larger caches, such as the `DataTagCache`, use the `BatchCacheLoaderDAO` to grab cache elements in batches on multiple threads, which are then loaded into the DB.



Cache loading vs cache module

The loading module retrieves the objects from the DB and passes these as a map to the cache module, which loads them into the cache.

**Modifying objects after DB retrieval**

Sometimes you will need to perform some modifications to objects retrieved from the DB before they are put in the cache. This can be placed in the abstract `doPostDbLoading` method in the `AbstractSimpleLoaderDAO` implementation. In particular, this is the best place if you need to use Spring properties to fill some fields.

**Post DB loading using several cache objects/caches**

More complex logic on filling cache objects - requiring access to other cache objects - should be placed in the cache implementation itself, not this module. See for example the rule parent id loading in `RuleTagCacheImpl`.

Cache persistence module

This module manages the persistence of cache updates to the database (only "data/value" updates are dealt with here, not updates to the configuration of the cache objects: these are handled by the [Configuration module](#)). The design follows the cache listener pattern 3 described here: [Writing a cache listener for a server cluster](#). This module registers listeners to all caches that need persisting and then saves the current values to the database. The registration is to receive the cache element keys only, in regular batches (using the `registerKeyBufferedListener` method).

The module includes a `CachePersistenceDAO` for persisting batches of updates to the DB in a single transaction. A listener must be instantiated for each cache that needs this functionality (`PersistenceSynchroListener`). It registers at start-up as cache listener to the appropriate cache.

**Asynchronous persistence**

As should be clear from the above description, the persistence of cache objects is done asynchronously. In particular, a server crash can lead to a discrepancy between server and cache (in the distributed cache setup), or missing data (in the local cache setup). The consistency is restored by asking for the latest updates from the DAQ and persisting the entire cache at start-up (recovery mode).

Handling persistence errors

If the persistence of some batch of cache elements fails, all these elements will be re-submitted at the next persistence attempt (also at server shutdown). The server will not shutdown correctly if the module has been unable to persist some objects.

The data source used is the same as that in the [Cache DB access module](#).

**Persisting cache objects manually**

This module also allows other modules to explicitly add a cache element to the list of those that need persisting. For this, the `BatchPersistenceManager` exposes a `addElementToPersist(Long id)` method. For example, this is used by the [LASER server module](#) once it updates the `isPublished` field.

Configuration module

The configuration module handles the runtime reconfiguration of the C2MON platform, including sending the required changes down to the DAQ layer. This logic behind the reconfiguration is perhaps the most complicated within the server, and for this reason the development was very much test-driven (only those scenarios contained in the tests should be fully trusted!).

Details of the reconfiguration functionality of the C2MON server have been included in the general architecture section [Server data configuration](#)

A single `ConfigurationLoader` implementation is exposed to the rest of the server and can be called to apply a given configuration (id is passed). The configuration module then retrieves the configuration from the database (3 tables: `TIMCONFIG`, `TIMCONFIGELT`, `TIMCONFIGVAL`).

The configuration module - and the server in general - have been designed to allow a single reconfiguration at a time: the server will only allow one configuration to run, and any new configuration request will be forced to wait until the previous is complete. The choice helped simplify the logic, since a number of cache actions can only take place during reconfiguration and do not need synchronizing across multiple threads (removal from the cache for example).

Database and cache consistency

In theory, the C2MON server attempts to keep the DB & cache in a consistent state, whatever the outcome of a configuration (including a server crash). In practice however, you may be faced with a crash leaving the loaded configuration in an inconsistent state. This arises because the cache does not use the same rollback mechanism as the database, meaning a sudden interruption in the configuration process can prevent the server from taking any action to restore the cache-DB integrity (for e.g. a frozen thread due to some locking problem). On the other hand, the design should provide a number of guarantees:

- If a configuration runs to completion, the cache & DB will be left in a consistent state (whether the configuration report indicates a

- success or not!)
- The configured elements on the database will be kept in a consistent state, meaning all create/update configuration elements will be either applied in their entirety or not at all, and all recursive remove elements will either be entirely applied or if interrupted will have interrupted at a well-defined point (see [Server data configuration](#) for details of the recursive removals). In particular, re-applying the removal will in most cases finish removing the element entirely, the only exception being control points.
- Cleaning the cache and restarting the server will re-synchronize the cache and DB, with the cache being loaded afresh from the database.



JConsole for quick recovery

To fix inconsistencies between the cache & DB, you can always restart the server after wiping the cache. On the other hand, the C2MON server exports cache methods via JMX to remove/load cache elements from the database manually. This can help avoid a server restart (assuming you have identified where the problem lies), including a Terracotta server clean restart if you are running in a distributed setup.

Alarm module

The alarm module evaluates internal C2MON alarms on changes to C2MON tags & rules. This should not be confused with the [LASER server module](#), which then publishes these alarms to the external LASER application. Alarms are also published to the C2MON clients for displaying to users.

Alarms can be attached to any Tag (datatag, control tag or rule): see for details. When a tag changes, the Alarm module must re-evaluate the alarm for this Tag. The main bean in the Alarm module is the AlarmAggregator, which takes care of integrating the alarm evaluation with the C2MON client module.

The AlarmAggregator

Tags are published with any associated alarms to C2MON clients, so a GUI application can display this information in parallel. To ensure the correct synchronization of this information, this module integrates Alarm evaluation with Tag publication: the AlarmAggregator listens to all updates to Tags, evaluates any alarms that may be attached, and notifies the client module of the Tag update (passing any attached Alarms at the same time). To achieve this, the client module registers as a *AlarmAggregatorListener* with the *AlarmAggregator*. Other modules that may be interested in Alarm changes should register directly with the Alarm cache (see f.eg. the *AlarmPublisher* in the [Client module](#)).



Dependency

This design introduces a dependency of the client module on the alarm module. This dependency is only required if tag & alarm publication need publishing to clients in a synchronous manner. A skeleton *AlarmEvaluator* class is provided for an alternative design, where Alarms are evaluated independently of Tag publication (and modules interested in Alarm updates would register directly with the *AlarmCache*).

LASER server module

The LASER module publishes internal C2MON alarms to the external LASER alarm system.



Dependency

This module requires the Alarm module to be installed also.

The LASER module performs to separate functions, that are implemented in the LaserPublisher and LaserBackupPublisher respectively:

- publish C2MON Alarm changes to LASER
- regularly publish a list of all active Alarms to LASER as backup

The Alarm object in the cache contains a "published" field that tracks whether an Alarm has been successfully published to LASER by the LaserPublisher. The latest published state is also stored. This allows a complete decoupling of the Alarm evaluation and publication to LASER (they run on separate threads and publication can be delayed indefinitely if the LASER publication interface is unavailable).



De-coupling from LASER

These features of the Alarm cache object currently allow for one publication tracking. Within CERN this is used for LASER publication, but in principle can be re-used to track publication to any Alarm system (the `isPublishedToLaser()` method is simply wrapping the `isPublished()` method)

The LASER publisher

The LaserPublisher follows the "Asynchronous listener with cache access" model describe in [Writing a cache listener for a server cluster](#). It is notified on its own thread of updates to the Alarm object, which is then retrieves, locks and publishes to LASER. The retrieval and locking ensure

the *latest* update is published, even if the original one generating the notification has been overwritten (which can happen since the module is notified on its own thread). Notice also that *only* alarms that have not already been published will be published again, so the listener can be notified repeatedly to restore consistency, without multiple publications occurring.

If LASER is unavailable for some reason and the publication fails, the publisher keeps track of this alarm for re-publication once LASER is back. A re-publication thread checks every 2 minutes if there are *local* unpublished alarms, and will attempt to re-publish them. This thread will re-attempt publication until successful. Notice the module only tracks these re-publications locally, but before publishing the alarm, checks again if the alarm has been set as published (internal shared state): in this case, the local re-publication thread will not publish the alarm.



Unpublished alarms on server shutdown!

A C2MON server will not shutdown smoothly if the LASER module has unpublished alarms, but will need killing. In this case, the server should be restarted in recovery mode to ensure all unpublished alarms are published (even if LASER is still not available). Alternatively, restart it normally and use the JConsole to call "publish un-published alarms".

The LASER backup publisher

The LASER system requires a regular re-publication of all active alarms: this LaserBackupPublisher is configured to re-publish them every 10 minutes (each server will do so). This functionality is provided by the LASER client API, but cannot be re-used for the C2MON server cluster setup, since members of the cluster will have a different sets of published alarms (an alarm change is published by a single server). Hence the backup mechanism needed implementing on the C2MON side.

To make sure Alarm changes are always published by the LaserPublisher first and that the backup only contains already-published alarms, it was unfortunately necessary to freeze all alarm publications during a backup creation and publication. What's more, the creation of a backup requires an transversal of *all* the alarm cache to collect the active ones. When the cache is not held in the local JVM (e.g. either on disk in the single server setup or on the Terracotta server/disk in a distributed setup), this process can be time consumer. For example, the TIM system will take around 20s to create this backup at server startup and after a GC, during which time no alarms will be published by any C2MON cluster member. To improve this design, the LASER API would need to have a more sophisticated backup mechanism allowing alarm sources of clustered applications with individual backups.



In the case of C2MON, the backup publication is to ensure integrity of the LASER system itself, since *all* latest alarm values are published via the LaserPublisher.

Client module

The client server module manages all communication with applications using the C2MON client API. It covers the following functionalities:

Tag subscription & publication Alarm subscription & publication Command reception & report Configuration request & report Communication via Jason & serialized messages

In general, the Google Gson implementation of the Json standard are used for communicating between the server and Client API. Serialized objects are only used for the command reception, since it is planned to send down the RBAC token in the future.

Java guidelines for C2MON server modules

All C2MON server modules follow a similar Java package structure. This serves 2 purposes:

- make modules easier to decrypt for future developpers
- provide a clear public interface to a module, which can be accessed by other modules

Each module has a single top-level package identical - or closely following - the name of the module. The package should always start with the common C2MON server top-level package: *cern.c2mon.server*. For example, the Configuration module has top level package *cern.c2mon.server.configuration*.

Spring configuration

All modules must use Spring as container, since the C2MON server requires a Spring XML file to load each module running in the server (these are listed in the *c2mon-modules.xml* file: see [Server configuration](#) for details). The module should provide a single top-level Spring XML file that can be imported by the server. This file should be named as *server-modulename.xml*, and placed in the *config* subpackage. For example, the Configuration module contains the *server-configuration.xml* file in the *cern.c2mon.server.configuration.config* package.

Exposed beans

A module may - or may not - choose to expose a number of Spring beans for use by other modules. These should be exposed through interfaces, and these interfaces should be placed in the top-level module Java package. Other interfaces may be placed in this top-level package for use by other modules.

For example, the Configuration module exposes the Configuration service through the ConfigurationLoader interface. The ConfigProgressMonitor interface is also exposed, as a user of the Configuration Loader service may need to provide one of these.

**Top-level package = Module public interface**

Only interfaces should be placed in the top level package! And the interface should be placed there if and only if it is destined for use by other module!

Maven artifact

The Maven artifact names should re-use the common group id used for all server module: *cern.c2mon.c2mon-server*. The artifact id should be the module name as used in the Java package. For example, the configuration module has artifact id *c2mon-server-configuration*.

Supervision module

Short-term-logging module

Server common module

The Common module contains structures shared across modules in the server, mainly the cache object interfaces and classes (see [Server cache objects](#)).

**Think carefully...**

... before adding classes to this package. Often they can be placed in the particular module that uses them. This project should contain mostly cache structures and server-wide constants (Enums).

Writing a new module

Many optional modules written for the C2MON platform will match one of the following two patterns, or a combination of both:

1. A cache listener pattern
2. A client request pattern

The cache listener pattern

The C2MON design encourages you to use a cache listener pattern to be notified of updates made to monitored points. To facilitate this, a number of registration calls are provided, both by the cache beans and the *CacheRegistrationService*. The registered listener can then take any required action on incoming data. More details on writing a C2MON cache listener can be found here: [Writing a cache listener for a server cluster](#).

**Module recovery after crash**

In addition to the usual callbacks when data is updated, a registered listener will receive callbacks when the server is attempting to restore the consistency of the system after a crash. In this case the callback is made on the *onStatusConfirmation* method. These callbacks may already have occurred via the standard channel, and the module must decide if it should take any action in this case or not.

An example of a module using this pattern is the Short-Term-Logging module, which logs Tag changes in a database.

The client request pattern

Modules may also need to receive and process requests from external systems, often sending a reply on completion. The provided modules that follow this pattern listen for requests on a JMS queue. The advantage of using JMS here is that it is directly compatible with a server cluster, with all servers listening for requests on the same queue. The request is then picked up by the "fastest" server (the exact JMS configuration can have an impact on which server picks up the request; see the section on optimizing your C2MON setup for these and related issues:).

In addition to these patterns, a module may choose to expose a service directly to other modules. For instance, the Short-Term-Logging module exposes a Command logging functionality so these can be manually logged when executed.

Lifecycle module

The lifecycle module contains the main C2MON server startup class called *ServerStartup*. The module manages:

- property loading into the server
- C2MON optional module loading
- C2MON cache datasource loading

Property loading

Property loading is specified in the *server-lifecycle.properties* file. The properties wherever they can be found first, from the following places:

1. Java properties
2. Specific property files (specified *c2mon.properties* file or the other standard files in the *conf* subdirectory: *c2mon-jms*, *c2mon-cache* and *c2mon-datasource.properties* files)
3. Default values kept in the *server-lifecycle.properties* file

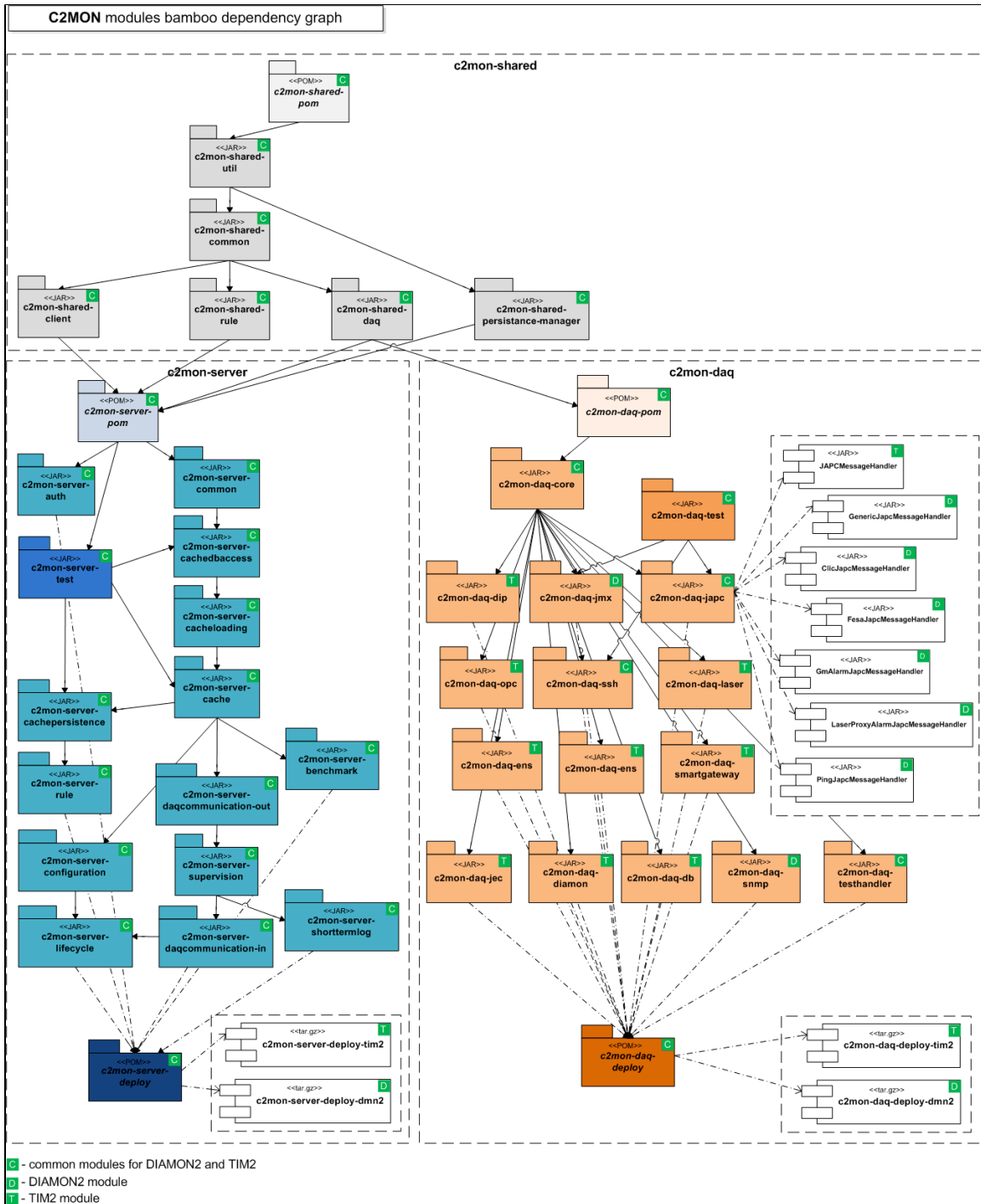


Core properties during testing

Notice this file is duplicated in the C2MON server test module, so the default properties can be used when running integration tests. If you add properties to any core modules, they will need adding to this file also for the tests to run.

C2MON Bamboo dependency graph

The C2MON modules are integrated with Maven, deployed to Nexus repository and automatically rebuild on build server (Bamboo) whenever some changes are committed. The build dependency of particular modules is presented on the following diagram:



Server configuration

This section describes how to configure a C2MON server.

Configuring the server modules

A C2MON server instance is made up of a core part and a number of optional modules. All core modules are loaded at startup and need to be present in the classpath. To include optional modules, the following 2 steps must be taken:

1. include the optional module in the classpath of your deployed C2MON instance
2. include the module top-level Spring configuration file in the `c2mon-modules.xml` file (using a Spring include statement)

That's it!



Module design and naming conventions

Don't forget to follow the C2MON coding conventions when adding a module: a single top-level Spring XML configuration file should be provided, following the standard naming convention described in [Java guidelines for C2MON server modules](#).

Overriding server properties

Many internal properties are available for configuring a particular instance of a C2MON server. These range from required database URLs & credentials to thread pool configurations and timeout values. Most of these are only used in Spring configuration files, a pattern we encourage to help locating where these properties are used. Default values are also provided for most values, so you only need to concentrate on those values you would like to customize.

To override a property, it can be included in one of the standard property files located in the server configuration directory:

- *c2mon-jms.properties* for JMS-related properties
- *c2mon-cache.properties* for cache-related properties
- *c2mon-datasource.properties* for DB-related properties

The location of the configuration directory is by default the *conf* directory in the C2MON home, which is specified at start-up with the Java option *-Dc2mon.home*. This property will then override the default value.

Another option is to specify required properties when starting the server using Java properties: *-Dproperty=value*. This will override any default setting or value set in one of the files above.

All properties available in the C2MON server are documented here: [List of server properties](#).



Adding new properties

First of all, only add a new property if customising the setting really makes sense! If you add a new property to some server module, add the default value to the appropriate file in the Lifecycle module: this enables to track all properties used in the server. To avoid clashes, use the trunk package name of your module as the trunk of property name.

Property loading takes place in the [Lifecycle module](#): see this page for details of where to add default values.

C2MON datasource

The C2MON core uses a single datasource, which needs including in any deployment of the server.



Re-using the provided C2MON datasource

If you are developing a new module, you may wish to re-use the provided cache datasource, assuming all your tables are kept in the same schema. For this, simply reference the Spring bean with id *cacheDataSource*, to inject it in your code.

List of server properties

This page documents all properties that can be set in the C2MON server, both for the core and provided optional modules. See also the individual module pages for further information.

Server core properties

Cache-related properties Datasource-related properties JMS-related properties

Optional module properties

LASER module properties Client module properties

Platform management

This section covers the management functionalities available when running a C2MON platform.

- [ActiveMQ broker management](#)
- [C2MON server management](#)
- [C2MON server recovery](#)
- [DAQ management](#)

ActiveMQ broker management

C2MON server management

C2MON server recovery

DAQ management

DAQ management is best performed through the ProcessState view, which gives details of the status of the DAQ processes. It also allows stops/starts of the Processes remotely.

Optimizing your C2MON setup

The C2MON platform offers numerous possibilities for customizing your setup. Most processes involved are highly configurable, including the C2MON server and data acquisition layer, as well as the JMS middleware and distributed cache settings. These settings include both "local" settings of the processes involved, as well as your overall choice of architecture. In this section we address these issues and how to approach them.



Hardware configuration

In this documentation we do not address the hardware configuration on which your C2MON platform is running. Of course, to maximize availability, all redundant processes should be placed judiciously on the different machines!

First of all, you should be asking yourself many questions about your requirements, as these can have major impacts on the final C2MON instance you choose to deploy:

1. Why I am monitoring?
2. What do I need to do with this data?
3. Is my data absolutely critical, or can I live if something is lost?
4. Can I partition my data in some logical way? In fact, does all my data need to be in the same system at all?

And a few indications as to why this might affect your setup:

Why I am monitoring?

- Am I doing real-time monitoring? If so, you probably have some desired maximum delay before data changes are displayed on the client. Or are you mainly interested in longer term analysis, in which case a longer delay may be acceptable during data avalanches. In this case you can probably settle for a simpler setup.
- Deciding what you need the data for will enable you to make good use of the filtering mechanisms available on the DAQ layer. This will greatly reduce the data throughput on the server level and allow you to run a smaller setup (and save resources!)

What is done with the data?

This is a key question, and needs precisely answering before discussing any performance issues. CPU will most certainly not be the limit in any C2MON installation: the limit will be reached on IO distribution or some other storage device.

- Does the data need logging? If so, does it all need logging? (logging can be switched off on the C2MON Tag level)
- How much of the data is a connected client likely to be subscribe to at one time? (this could be limited for instance)

Is my data critical?

- For instance, you may have alarms based on it, in which case any action must be guaranteed. This has implications on your JMS setup, which should probably be transacted at the data acquisition level and persistent on the broker.

Can I partition my data?

This means, can I separate my data into distinct sets in some logical way. For example, can I split the data so no C2MON rules will even make use of data from 2 distinct sets. This has major implications for the scalability of the system, since partitioning data on the DAQ layer along these lines will enhance the distributed cache performance. In fact, it may be more appropriate to manage sets of your data on different C2MON instances altogether: see the remark below for merging this data again on the client layer.

**Connect a client to several C2MON instances**

This is not yet implemented but already planned in the design: a single C2MON client will have the possibility to register for data from completely separate C2MON instances. For this, a URI description is used in the client application for uniquely identifying the origin of the data. What's more, a user can then combine this data locally using client rules.

Some concrete architecture examples

To make this discussion more concrete, we present a few deployment scenarios for addressing typical needs, with a simple list of associated requirements. Of course, these scenarios can be combined to provide mid-way solutions.

**Beware of numbers!**

The numbers below are given only as an indication, and assume you are using all the available modules. In particular, this includes database logging and distribution to clients of all these values. Note also this is not the amount of data arriving at the data acquisition layer, which can be orders of magnitude higher (due to the filtering strategies).

Architecture scenario 1: moderate size of monitored data (<500'000 monitored points); high throughput (>1000 point updates per second); acceptable planned short service interruptions; availability not critical; low budget.

Architecture scenario 2: moderate size of monitored data; average throughput (<300 updates per second); absolute minimum of service interruptions; high availability requirements; low budget

Architecture scenario 3: very large amounts of data; high throughput; absolute minimum of service interruptions; high availability requirements; low budget (!?)

As examples, the DIAMON and dTIM projects at CERN closely follow the first and second architecture respectively.

**Further optimization**

Even if the settings discussed here do not meet your needs, it may be a small change in some of the C2MON components can! Many small changes are possible to balance performance and reliability concerns.

Tuning the distributed cache

Tuning the distributed Terracotta cache is an art in itself, and we recommend you study the (very good and detailed!) documentation on the Terracotta site (open source documentation). In particular, depending on your network performance and stability, you will need to tune the timeouts for nodes leaving the cluster and rejoin attempts!

Tuning your ActiveMQ brokers

Again, this is an art in itself. Best advice is to make sure you test platform is a mirror of your production one: this will help you spot any problems early enough. Make plenty of tests on broker/server restarts to check the setup is robust enough.

Architecture scenario 1

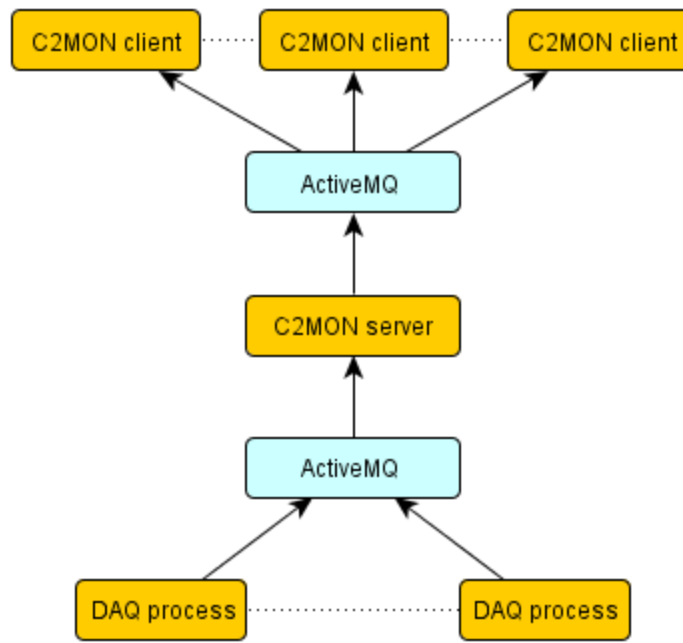
Requirements: moderate size of monitored data (<500'000 monitored points); high throughput (>1000 point updates per second); acceptable planned short service interruptions; availability not critical; low budget

For such a scenario, probably a single-server configuration is sufficient. You lose the redundancy provided by a server cluster, as well as the advantage of rolling updates. On the other hand, the in-memory cache immediately provides a very good performance and your setup becomes simpler and easier to manage.

**Changing C2MON architectures**

It is always possible to switch to another architecture at a later stage, if you decide you need the advantages of server redundancy or need to scale up.

The system layout is given below. The ActiveMQ boxes can either represent a single or distinct brokers (and can of course be replaced by clusters).



Ehcache overflow to disk

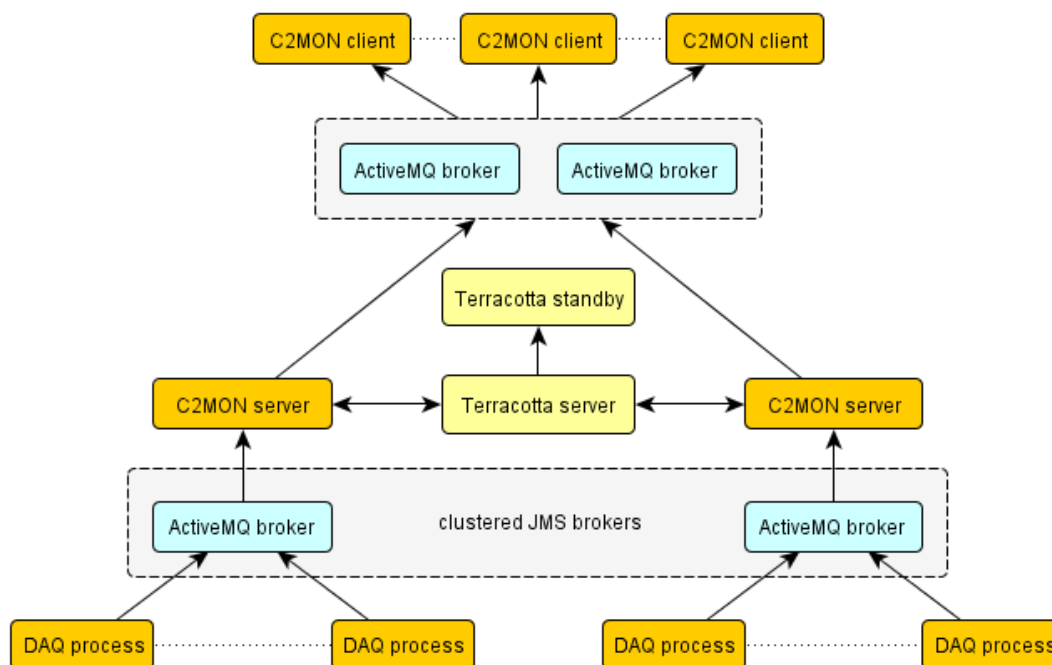
If the size of your cache starts influencing performance (usually at GC time) or is too large for the memory heap available, you could try limiting the in-memory size of the cache and overflowing to disk (will work best if some of your data is rarely accessed/updated). See [Ehcache](#) for details.

Architecture scenario 2

Requirements: moderate size of monitored data; average throughput (<300 updates per second); absolute minimum of service interruptions; high availability requirements; low budget

The architecture presented below combines high availability and maintainability with sufficient performance for most needs. As compared to [Architecture scenario 1](#), it has added redundancy on the server level, allowing rolling updates and instant failover in case of a single server failure. With the separate Terracotta caching process, this architecture choice also allows for a larger number of configured data points out-of-the-box, since the Terracotta process will automatically overflow to disk if required (on top of providing an extra JVM heap for data storage alone). On the other hand, the performance will in general be a slower and allow less throughput, since the transfer of cache changes to Terracotta is transacted (including the disk cache persistence).

The diagram below gives the layout in this example.



Architecture scenario 3

This is the ultimate scenario with the generalized high requirements. Of course, if you really want all this, the "low budget" part will need compromising, although it may not be as expensive as you may imagine (just no longer free). On the other hand, most cases don't have these high requirements: what you assume to be a high requirement in terms of performance for example, may not be so high these days if you have the correct underlying hardware. Even more importantly, it may be a re-think of your data, what you want out of it, and how you propagate and filter this on the data acquisition level for instance, can substantially reduce your original requirements estimate, so simpler configurations may be options after all.



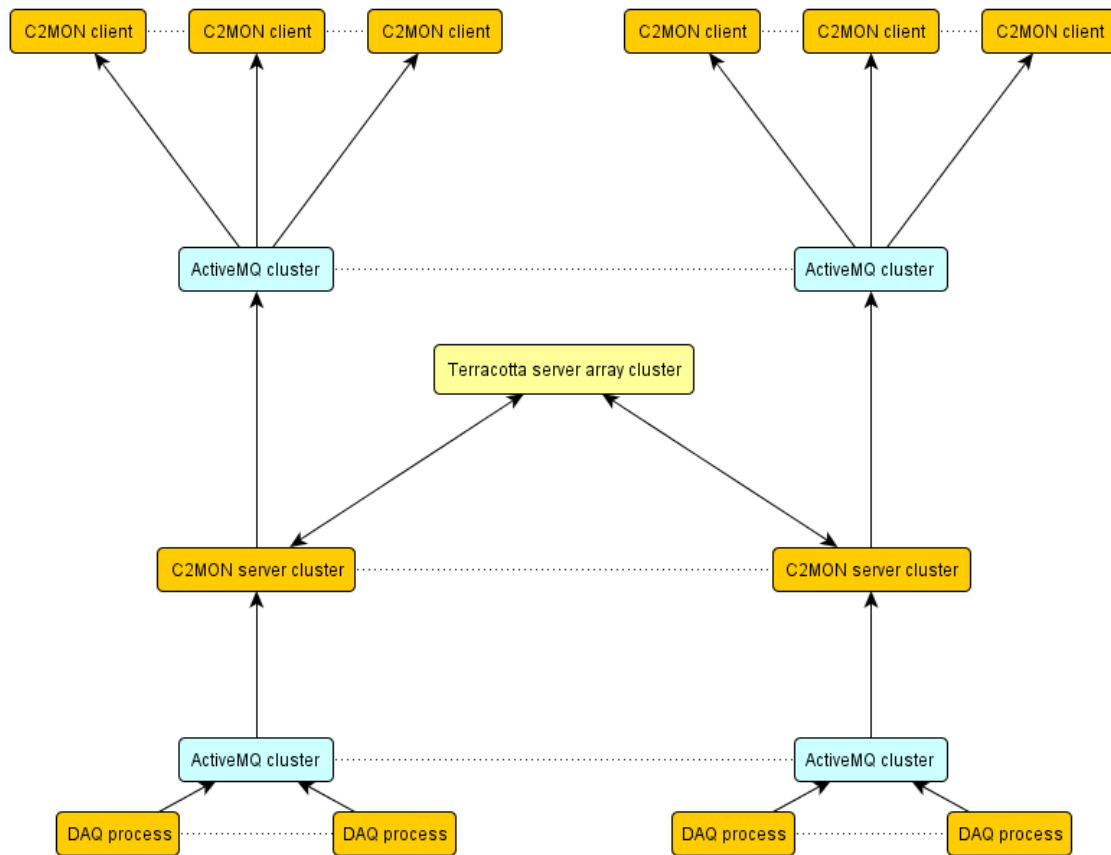
Separate C2MON instances?

Don't forget, if your data splits into completely independent sets you may be better off running several C2MON instances!

That said, the C2MON platform is designed to scale at all levels, so if you really need this, you can get it. We list below the choices you would make to build the "dream" C2MON deployment. First, however, you should partition your data: if you have so much data, surely it can be broken down in some way, with minimal dependencies between the partitions.

1. On the DAQ layer, dedicate a single DAQ to a given data partition. Make the partitions small enough so that the DAQ can handle the load. Create as many DAQs as this requires.
2. The JMS middleware: run a collection of ActiveMQ clusters, with each cluster dedicated to a fraction of the data partitions. A DAQ only connects to the cluster dedicated to its data partitions.

Below is the high level architecture overview. Note many nodes are in fact clusters, typically with two internal components for redundancy. The diagram only displays a couple of horizontal data partitions on the central level, but all levels can be horizontally scaled along the dotted lines. Notice each lower part of the tree consisting of a number of DAQ processes, a JMS and C2MON server cluster, should be configured as described in [Architecture scenario 2](#), to optimize the cache performance.



Terracotta license fee

The Terracotta server array setup used above is subject to a license fee. For limited data amounts, the current setup could be used without this option.

Client Applications

Content

- [C2MON Client API](#)
- [C2MON Publishers](#)
- [C2MON Test Client](#)

C2MON Client API

The C²MON client API is the API exposed to client applications that wish to interact with a C²MON server (at the same time, we often use this terminology for the implementation of the API).

Configuring the Client API

For configuring the size of tag requests to the server see



[TIMS-669](#) - Adjust client tag request to accomodate very large request sizes (Resolved)

C2MON Publishers

Data Publisher Documentation

C²MON provides to external publisher which are based on the C²MON Client API. The purpose of these publishers is to provide data gathered with C²MON to other parties via CERN standard middle-ware protocols.

- [DIP Publisher](#)
- [LEMON Publisher](#)
- [RDA Publisher](#)

Creating a new C²MON publisher from scratch

Creating a new independent C²MON publisher that converts data into whatever format you want is quite simple. All you have to do is to implement the [Publisher](#) interface which is provided by the [c2mon-publisher-core](#) package and to annotate your class as [SPRING @Service](#). The publisher core will handle for you the data subscription in the background. All you have to do is to provide a list of tag ids to which your publisher has to subscribe.

Coding guideline

The following guideline explains you the main steps that you have to do to create your own C²MON publisher:

- Create a new Maven project and add the [c2mon-publisher-core](#) artifact as dependency. The [c2mon-publisher-core](#) contains beside the [Publisher](#) interface also the [PublisherKernel](#) start-up class which contains the `main(String[])` method for launching later your publisher.

```
<dependency>
  <groupId>cern.c2mon.c2mon-publisher</groupId>
  <artifactId>c2mon-publisher-core</artifactId>
</dependency>
```

- Your publisher code has to be located in a sub-package of `cern.c2mon.publisher` so that SPRING is able to find and scan your class for annotations.

```
package cern.c2mon.publisher.foo;

...
```

However, if for any reason you do not want to put your publisher into a `cern.c2mon.publisher` sub-package you have then alternatively to define your own SPRING configuration. Otherwise the publisher core is not able to find your [Publisher](#) interface implementation! The location of your SPRING configuration can be specified by the following Java environment variable and will be added to the same context as the core services: `-Dc2mon.publisher.spring.configuration.location=...`

- Annotate your class as [SPRING @Service](#). This will assure that your class is instantiated a Singleton instance by the SPRING container.

```
@Service
public class PublisherSample implements Publisher {
  ...
}
```

- Make sure that your publisher class implements the [Publisher](#) interface which is provided by the [c2mon-publisher-core](#) package. The publisher core package will then handle the subscription to your tags in the background and will inform you about every event through the `onUpdate()` method.
- For a correct start-up of the [PublisherKernel](#) Main class you have at least to provide the following environment variables:

| Java environment variable | Mandatory? | Description |
|---|------------|--|
| log4j.configuration | YES | Location of the Log4j configuration file |
| c2mon.client.conf.url | YES | URI of the C ² MON client properties. This depends for which system you are writing the publisher (TIM or DIAMON) |
| c2mon.publisher.tid.location | YES | Location of the TID configuration file that contains the list of tag IDs to which the publisher shall subscribe |
| c2mon.publisher.spring.configuration.location | NO | This environment variable is optional and can be set to point to another Spring context file that should also be scanned at start-up and included in to the same context |



Info

Please notice that we use the SPRING 3.0.x framework for C2MON. If, you want also to use SPRING for your project, please make sure that you are using the same version as defined in the [c2mon-client.pom](#) !

Hello World example of a C²MON publisher

PublisherSample.java

```
package cern.c2mon.publisher.sample;

import org.springframework.stereotype.Service;

import cern.c2mon.client.common.tag.ClientDataTagValue;
import cern.c2mon.publisher.Publisher;
import cern.c2mon.shared.client.tag.TagConfig;

@Service
public class PublisherSample implements Publisher {

    /**
     * This method is called every time a value update is received
     * @param valueUpdate The tag value update.
     * @param tagConfig the related tag configuration which provides useful
     *                  additional static information about the tag.
     */
    @Override
    public void onUpdate(final ClientDataTagValue valueUpdate, final TagConfig tagConfig) {
        // TODO: add here your publication code!

        System.out.println("Received update for tag " + valueUpdate.getId());
    }

    /**
     * This method is called before shutting down the publisher instance
     */
    @Override
    public void shutdown() {
        // TODO: Handle here the publisher shutdown

        System.out.println("Bye!");
    }
}
```

DIP Publisher

| | |
|--------------|------------------------------|
| Developed by | Matthias Braeger, GS-ASE-SSE |
| CERN Contact | TIM Support |

Overview

The current implementation of the DIP publisher for C²MON is an external application based on the C²MON Client API. It subscribes to preconfigured data tags and publishes the values on DIP. By default the TIM tags are not published via DIP. This is only done on [request](#).

The current version of DIP Publisher works with **DIP v.5.4.4**.

The data tags are published as DIP topics with the following field names which are similar to those of the [RDA Publisher](#):

| DIP Field Name | Field type | Description |
|----------------|------------|---|
| id | long | A unique ID of the property which corresponds to the tag id within C ² MON. This id is for instance useful, if you want to check the same property with the TIM Viewer application |

| | | |
|-------------------------|--|---|
| value | float, double, integer, long, short, String, boolean | The data tag update value |
| valueDescription | String | Further information to the value, if specified by the source |
| timestamp | long | A continuous growing timestamp (number of milliseconds since January 1, 1970, 00:00:00 GMT) which is added by the C ² MON server when the property is inserted to its cache. |
| sourceTimestamp | long | The source timestamp (number of milliseconds since January 1, 1970, 00:00:00 GMT) of the value which is generated by the equipment when sending a value update. Please notice that this timestamp can also jump back in time, which might happen for some properties e.g. when C ² MON loses temporarily the connection to the equipment. |
| unit | String | The unit of the measurement, e.g. <i>bar(a)</i> |
| name | String | The name of the data tag |
| description | String | A general description of this data tag |
| mode | String | Is always either OPERATIONAL, TEST or MAINTENANCE |
| simulated | boolean | true, if the value is currently simulated and not really coming from an equipment |
| wiki | String | The URL to this wiki page |
| pointDetails | String | The URL of the HelpAlarm page of the published data tag. HelpAlarm provides many more details about the given point, for example the responsible person, GMAO code, attached Alarms, etc. |

TIM Publications

All DIP publications which are coming from the TIM service are published under the following DIP folder:

| | |
|----------------|------------------|
| Example | dip/ts/TIM/xxxxx |
|----------------|------------------|

Please follow this [link](#), if you want to know more about how to request/subscribe to DIP publications for the Technical Infrastructure Monitoring (TIM) project:

<http://wikis/display/TIM/TIM+DIP+Publisher>

How to find the responsible person for a DIP publication?

Finding the responsible person or group for a given DIP publication is often not so easy. The DIP homepage provides for this purpose a [Contact List](#) which should help you getting in touch with the responsables.

Useful Links

- [DIP Homepage](#)
- [DIP Browser](#)
- [DIPG Contact List](#)
- [TIM-DIP-Publisher Problem Solving Guideline](#)
- [Source Code](#)

RDA Publisher

| | |
|---------------------|------------------------------|
| Developed by | Matthias Braeger, GS-ASE-SSE |
| CERN contact | C2MON Support |

Overview

The [RDA](#) publisher for C²MON data tags is an external publisher that is built on top of the C²MON Client API. It enables to send any **non-redundant** value updates through [RDA](#). **By default the C²MON data are not sent through RDA**. For the TIM project the publication has first to be requested with a [MODESTI](#) request. This procedure assures that the publication is not accidentally removed without first having consulted the initiator.

Please notice that C²MON is NOT publishing data on a regular basis but only if there is a real value change.

The data tags are published as RDA properties with the following field names:

| RDA Field Name | Field type | Description |
|--------------------|--|---|
| id | long | A unique ID of the property which corresponds to the tag id within C ² MON. This id is for instance useful, if you want to check the same property with the TIM Viewer application |
| value | float, double, integer, long, short, String, boolean | The data tag update value |
| valueDescription | String | Further information to the value, if specified by the source |
| timestamp | double | A continuous growing timestamp (number of milliseconds since January 1, 1970, 00:00:00 GMT) which is added by the C ² MON server when the property is inserted to its cache. Please notice that this timestamp is sent as double field in order to be compatible with JAPC. |
| sourceTimestamp | long | The source timestamp (number of milliseconds since January 1, 1970, 00:00:00 GMT) of the value which is generated by the equipment when sending a value update. Please notice that this timestamp can also jump back in time, which might happen for some properties e.g. when C ² MON loses temporarily the connection to the equipment. |
| unit | String | The unit of the measurement, e.g. bar(a) |
| valid | boolean | true, if the quality of the tag is good (OK). THIS FIELD SHOULD ALWAYS BE CHECKED IN ADDITION TO KNOW WHETHER YOU CAN TRUST THE VALUE! |
| quality | String | Is always OK for valid tags, otherwise it can be set to one of the following error codes: UNINITIALISED, INACCESSIBLE, VALUE_EXPIRED, VALUE_OUT_OF_BOUNDS, INVALID_TAG, VALUE_UNDEFINED, UNKNOWN. It is also possible that you receive a concatenation of several error codes, e.g.: VALUE_OUT_OF_BOUNDS+INACCESSIBLE Please notice that this field is more for advanced usage. Normally it is enough to check, if the property is valid (see valid field description). |
| qualityDescription | String | Further information concerning the quality of the property |
| name | String | The name of the data tag |
| description | String | A general description of this data tag |
| mode | String | Is always either OPERATIONAL, TEST or MAINTENANCE |
| simulated | boolean | true, if the value is currently simulated and not really coming from an equipment |

C²MON RDA device names

The following table shows which devices are actually defined. Each device name corresponds to one C²MON RDA publisher instance.

| Device Name | Description |
|----------------|--|
| TIM.RDA.DEVICE | The RDA publisher for the Technical Infrastructure Monitoring (TIM) system |
| DMN.RDA.DEVICE | The RDA publisher for the Diagnostics and Monitoring (DIAMON2) system |

Reading RDA values through JAPC

A JAPC parameter name is usually given by the following triplet:

```
device/property(#field)
```

The **property** name for values coming from a C²MON RDA publisher corresponds to the unique tag name within the system (TIIM or DIAMON).

Useful Links

- [MODESTI request](#)
- [JAPC homepage](#)

- [RDA homepage](#)
- [Source Code](#)

LEMON Publisher

Overview

The LEMON publisher for C²MON is an external application based on the C²MON Client API. It subscribes to preconfigured data tags and publishes the values on LEMON

C2MON Test Client



C2MON Test Client

A test client can be found in the c2mon-client-test project. It is ready for deployment on TIM machines & for TIM datatags. For other systems, the deployment descriptor would need modifying (see deployment.xml).

The main class is C2monTestClient and it can be started using the provided c2mon-test-client.sh script in the bin directory (it takes the number of clients to start as argument). Another script is provided to kill all these clients.

A client will connect to a random selection of tags listed in a text file (script fixes the number of tags to 1000).

See the following issue:

 **TIMS-387** - Create test client application for connecting many clients to test system ( Resolved)