

Notes on the Monitoring and Information Service

Mariano A. Zimmler

July 26, 2004

Python client API

This document specifies the usage of the python client API for the Monitoring and Information Service. Other than syntactics, it is the same as the C++ client API.

1. Module imports

```
import MonitoringEvent
import MonitoringEventHandler
```

2. Create an event

```
event = MonitoringEvent.MonitoringEvent()
```

3. Add data to the event

The base event class MonitoringEvent implements the usual methods of dictionaries. Thus, to add data to it any of the following may be used:

```
event[MonitoringEvent.TYPE_TAG] = "theEventType"
event[MonitoringEvent.ID_TAG] = "theEventId"
event.addData(MonitoringEvent.PARENT_ID_TAG, "theParentEventId")
event.addData(MonitoringEvent.PRODUCER_ID_TAG, "theEventProducerId")
```

Subdictionaries may be added in the same way as ordinary data:

```
dict = {}
dict["Attribute_1"] = "theFirstAttribute"
dict["Attribute_2"] = "theSecondAttribute"
event.addData("DictionaryAttribute", dict)
```

It is also possible to append dictionaries to the data already stored in the internal event dictionary:

```
additionalData = {}
additionalData["ExtraAttribute_1"] = "theFirstExtraAttribute"
additionalData["ExtraAttribute_2"] = "theSecondExtraAttribute"
event.addDict(additionalData)
```

4. Get the event handler instance

The MonitoringEventHandler instance handles the communication with the server through the ServerProxyManager class. This class is thread safe and is responsible for verifying that the server is available and sending the event.

```
eventHandler = MonitoringEventHandler.getInstance()
```

5. Send the event

In the current implementation, the `processEvent` method checks whether the server is available every time it is going to send an event. If the server is not available it simply writes such a message to the log file and returns.

```
eventHandler.processEvent(event)
```

6. Complete class diagram for the base event class

6.1 Constants

```
UNKNOWN_STRING_VALUE = "__UNKNOWN__"  
GENERIC_TYPE = "GenericEvent"  
ID_DELIMITER = "__"  
  
NAME_TAG = "EventName"  
TYPE_TAG = "EventType"  
ID_TAG = "EventId"  
PARENT_ID_TAG = "ParentEventId"  
PRODUCER_ID_TAG = "EventProducerId"  
DESCRIPTION_TAG = "EventDescription"  
TIME_STAMP_TAG = "TimeStamp"
```

6.2 Exception classes

```
class EventException(MisException.MisException)  
class UnknownValue(EventException)  
class UnknownTag(EventException)  
class ProcessingError(EventException)
```

6.3 MonitoringEvent class

```
def __init__()  
def getUniqueId(eventInitials, producerId)  
def getType()  
def getId()  
def getParentId()  
def getProducerId()  
def getDescription()  
def getTimeStamp()  
def getDictionary()  
def getStringRep()  
def getXMLRep()  
def getEventDescription()  
def addData(key, value)  
def __setitem__(key, value)  
def addDict(dict)  
def addEventDescription(eventDesc)  
def __getitem__(key)  
def get(key)
```

This base class implements the usual accessor methods of dictionaries. Thus, data that has been stored in the internal dictionary may be accessed in several ways. Examples follow.

```
print TYPE_TAG + ": " + event.getType()  
print ID_TAG + ": " + event.getId()  
print PARENT_ID_TAG + ": " + event.getParentId()
```

```
print PRODUCER_ID_TAG + ": " + event.getProducerId()
print "Unique id: " + event.getUniqueId("TEI", event.getProducerId())
print "ExtraAttribute_1: " + event["ExtraAttribute_1"]
print "ExtraAttribute_2: " + event.get("ExtraAttribute_2")
```

It is also possible to retrieve the entire dictionary as well as a nice XML representation of it:

```
print "Event data:\n" + `event.getDictionary()`
print "String representation:\n" + event.getStringRep()
```

This base class provides all the necessary methods for converting back and forth between python dictionaries and the IDL type `eventDescription`, which is what the server takes in. Thus, this class can be instantiated as an event comes into the server and used to store the original event dictionary as well as the unpacked data.