LBT–ADOPT
TECHNICAL REPORT

Doc.No : LBT-AdOpt.nnn
Version : 1.0
Date : dd Mmm yyyy

AO-Arbitrator: requirements analysis

Fabio Tosetti

INAF – Osservatorio Astrofisico di Arcetri
Largo E. Fermi, 5 - 50125 Firenze - ITALY
http://www.arcetri.astro.it/adopt/

ABSTRACT

This paper describes the functional and not-functional requirements for the AO-Arbitrator module. Moreover is defined a possible AO-Arbitrator architecture deriving from these requirements and from the AO System general requirements, current functionalities and architecture.

Modification Record

Version	Date	Author	Section/Paragraph affected	Reason/Remarks
1.0	17 Oct 2007	Fabio Tosetti		First release of the document

Abbreviations, acronyms and symbols

Symbol	Description
LBT	Large Binocular Telescope
Arbitrator	Generic arbitrator module, that is a component designed to coordinate some tasks.
AO-Arbitrator	Top-level arbitrator for the AO system: it controles the WFS-Arbitrator and the AdSec-Arbitrator.
WFS-Arbitrator	Arbitrator for the WFS subsystem.
AdSec-Arbitrator	Arbitrator for the adaptive secondary subsystem.

Contents

1 Introduction

The AO-Arbitrator is a module of the AO-Supervisor designed to execute the set of commands required for the AO-Supervisor OBSERVATION mode[1] defined by the AOS [1].
This module is intended to coordinate all the AO-Supervisor internal tasks necessary to perform the high-level interaction with the Adaptive Optics system: the high-level interface includes all the commands exported from AOS to TCS environment, and a set of calibration features implemented inside the AOS (but not exported to TCS)[2]. It is not intended to manage any AO-Supervisor low-level operation required in STANDALONE or ENGINEERING mode.

In this discussion the AO-Arbitrator is considered as the only AO-Supervisor module which communicates with the AOS: therefore the name "AO-Arbitrator" will be always used instead of "AO-Supervisor" to indicate this side of the systems. There is also another reason for that choice: because the AOS module actually belongs to both worlds - the TCS and the AO-Supervisor - in principle it wouldn't be correct to talk about a communication between AOS and AO-Supervisor.

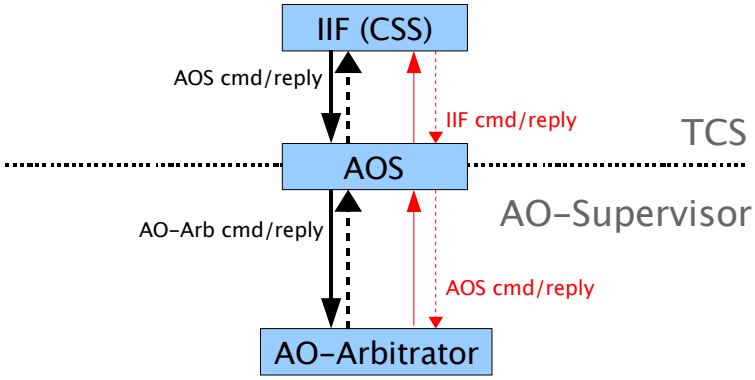

Figure 1: Commands from AOS to AO-Supervisor

1.1 System architecture guidelines

The f*igure 1* shows the high-level interaction from the AOS to the AO-Arbitrator, as defined in [1], paragraph 2.1: the thicker black arrows labeled "*AOS cmd/reply*" and "*AO-Arb cmd/reply*" represents this top-down communication. All details concerning the communication are deliberately ignored, because they don't have any implication on the AO-Arbitrator functional requirements, and by consequence on the AO-Arbitrator architecture.

It's important to note that there is also a communication in the opposite direction (from the AO-Supervisor to the TCS; [1], paragraph 2.2) implemented by the AOS to provide some TCS services[3] to the AO-Supervisor: the thin red arrows labeled "*AOS cmd/reply*" and "IIF *cmd/reply*" represent this bottom-up interaction.

[1] The CALIBRATION mode is at the moment considered as a special kind of OBSERVATION.
[2] Still tbd.
[3]Because the AOS is a subsystem of the TCS, all the services provided to AO-Arbitrator can be considered TCS services, also if they don't require any interaction with others TCS subsystems. This is for example the case of the logging commands.

In OBSERVATION mode the latest kind of communication admits only *Warning*, *Error and Panic* events, and some logging commands. The logging commands are defined in [1], and only *LogItem* should be used by the AO-Arbitrator to request a log into the TCS system log. The *Warning*, *Error* and *Panic* are here considered not just as commands to AOS, but as *events* reported by the AO-Arbitrator to the AOS to inform the TCS of some problem occurring during the AO functioning.
Summing up, the communication between the AOS and the AO-Arbitrator is composed by:

- Commands from AOS to AO-Arbitrator, with synchronous reply.
- Asynchronous events from AO-Arbitrator to AOS.
- Limited set of commands (logging only?!) from AO-Arbitrator to AOS.

On the other side, just inside the AO-Supervisor system, the AO-Arbitrator should communicate only with the two lower-level arbitrators, the *WFS-Arbitrator* and *AdSec-Arbitrator* (named WFS and AdSec sequencer in [1]).

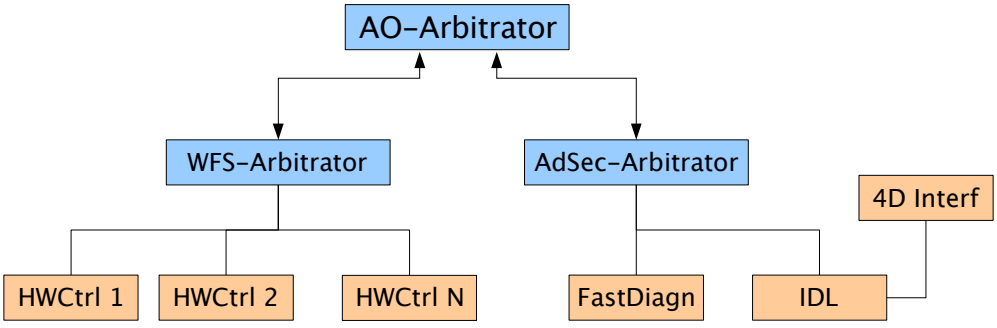

Figure 2: Interaction between AO-Arbitrator and others AO-Supervisor modules

As shown in *figure 2*, the communication of the AO-Arbitrator with the others AO-Supervisor modules must be strictly limited to the WFS-Arbitrator and AdSec-Arbitrator: this is a main architecture guideline and isn't discussed here.

1.2 Existing system modules

This paragraph sums up a list of the existing components of the AO system (AO-Supervisor + AOS), analyzing their current architecture and implementation and its impact on the AO-Arbitrator functionalities and architecture. In some cases, an alternative architecture or implementation is proposed to better match the system functionalities and/or the AO-Arbitrator requirements.
A **main system property** to be taken into account is the communication infrastructure, based on the MsgD-RTDB: all modules communicate using MsgD messages or variables.

1.2.1 AOS

The AOS is defined in [1].

1.2.2 WFS Arbitrator

Is implemented as a Python process (derived from python AOApp) using a set of Python hw-modules (see below) to control the Hardware Controllers.

Advantages:

- ✔ The Python hw-modules are also useful for testing and debugging scripts.
- ✔ The Python hw-modules act as a library, because they entirely take care of the communication with the Hardware Controllers.
- ✔ The main process can easily catch the warnings and errors rising up from the Hardware Controllers using the language exceptions generated by the python hw-modules.

Disadvantages:

- ✗ Must wrap external C libraries to use them, and force these libraries to be pure C, not C++.
- ✗ The Python hw-modules aren't easily integrable with the existing engineering GUI, which duplicates a lot of code to interact with the Hardware Controllers.

Probably this arbitrator should be maintained in its current implementation.

1.2.3 AdSecArbitrator

Implemented as a Python process (derived from python AOApp), that at the moment does almost nothing...
It must communicates with:

- FastDiagnostic (C++ application)
- AdSec controller (IDL process)

Could be rewritten using the AO-Arbitrator template (see below) to get all its features 'for free'. In this case, because the WFS-Arbitrator is python, should be evaluated the lack of symmetry of the system and its consequences.

1.2.4 Hardware controllers

Derived from the AOApp class written in C++ and handled using the Python hw-modules. This is the final implementation.

1.2.5 Fast Diagnostic

Derived from the AOApp class written in C++ and controlled using a C++ library (also exported to IDL). This is the final implementation.

2 Requirements

The AO-Arbitrator features are divided in *functional* and *not-functional* requirements. Moreover, the functional requirements are divided in three sections. The first section consider the needing for a general *Arbitrator Framework*:

- The AO-Arbitrator must be based on a A*rbitrator Framework,* suitable for:
  - Future extensions and modifications.
  - Reuse in the implementation of the lower-level arbitrators.

  Respect to the framework, the nature of the components *using it* (arbitrator clients[4]) and *used by it* (arbitrator servers[5]), doesn't matter. See below the *AO-Arbitrator Framework* requirement 2.

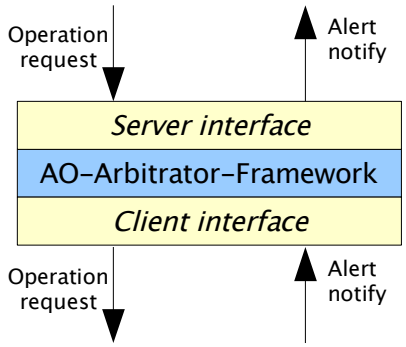

Figure 3: AO-Arbitrator-Framework interfaces

In *figure 3* is possible to note that an *AO-Arbitrator-Framework* can be chained with another AO-Arbitrator-Framework, because their interfaces are symmetric.
The other two sections of the functional requirements describes the functionalities required to the framework instance corresponding to the to actual AO-Arbitrator, that is the interaction with the AOS and with the lower-level arbitrators.

2.1 Functional requirements

These requirements are related to the interaction of the AO-Arbitrator with the *client components*[6], and are considered in term of services required from - and provided to - the external modules.

2.1.1 AO-Arbitrator Framework

The core of the AO-Arbitrator must be a framework defining the basic functionalities.

In the first set of requirements the *client* of the arbitrator is the module using it to implement a specific AO-Arbitrator: from this point of view the following are functional requirements.

[4] The AOS when the framework is used to implement the AO-Arbitrator.
[5] The WFSArb and the AdSecArb when the framework is used to implement the AO-Arbitrator.
[6] Here "client component" can have different meanings. See above.

1 The framework must define a FSM (Finite State Machine), where the state transition are triggered by external *commands* and *asynchronous events*.

1.1 The FSM defines *normal* states and *failure* states.

1.1.1 The failure states can be recoverable or not-recoverable.

1.1.1.1 Only a recoverable failure state allow to reach a normal state in the following transaction.

1.1.1.2 A not-recoverable state allow only emergency operations (like shutdown the system, *tbd*).

1.2 An external command generates a *state transaction* of the FSM.

1.3 An external command always generates a synchronous reply at the end of its execution.

1.3.1 The reply can be SUCCESS, WARNING or ERROR

1.3.1.1 Reply SUCCESS: the FSM has successfully reached the new state

1.3.1.2 Reply WARNING: the FSM has reached the new state but something unexpected – but automatically recovered - must be reported.

1.3.1.3 Reply ERROR: the FSM encounter an error while changing status, and reach a failure state.

1.4 An asynchronous event generates a state transaction or a transaction interruption.

1.4.1 A transaction interruption force the executing command to terminate and to return a WARNING or ERROR.

1.4.2 The result of a transaction interruption is a failure state.

The second set of requirements define the interaction with the external components.

2 The framework must provide 2 interfaces: a server-interface and a client-interface. Both the interfaces contributes to the status change of the FSM.

2.1 The server-interface exports a set of operations (“Operation request” in figure 3) and provide a feedback to clients.

2.1.1 An operation can result in a *command* or in an *asynchronous event* for the FSM.

2.1.2 The feedback can be synchronous or asynchronous.

2.1.2.1 A synchronous feedback is returned at the end of an operation execution as command result.

2.1.2.2 An asynchronous feedback (“Alert notify” in figure 3) can be returned during the operation execution or independently from a client request.

2.2 The client-interface handle all the services requested by the framework to external modules (servers).

2.2.1 These services includes outgoing operation request.

2.2.2 These services also include the management of incoming feedback from the servers.

2.2.2.1 Synchronous feedback is typically the result of an operation request.

2.2.2.2 Asynchronous feedback results in *asynchronous event* for the FSM.

2.1.2 AOS interaction

The communication with the AOS, in terms of MsgD communication, is defined in [1].

1. The AO-Arbitrator must provide an interface (*AOArbitratorLib*) to AOS, in order to abstract the provided services from the implementation (actually implemented as messages exchange or shared RTDB variables).
2. The AOS must provide an interface (*AOSLib*) to AO-Arbitrator, in order to abstract the provided services from the implementation (actually implemented as messages exchange or shared RTDB variables). The AOS services includes the alerts notify (*Warning*, *Error* and *Panic*) to TCS and the logging command (*LogItem*). Some extra services can be defined in [1].
3. ...

2.1.3 Wfs-Arbitrator and AdSec-Arbitrator interaction

[Tbd]

2.2 Not-Functional requirements

These requirements concern only internal factors, and so doesn't have any impact on the system architecture.

2.2.1 AO-Arbitrator Framework

– The AO-arbitrator must derive the C++ AOApp class, because it provides reliable and well-tested services to communicates with the MsgD.

– The FSM should be designed using the *Fsme* tool (Finite State Machine Editor), which provides an useful graphical representation of the finite state machine and an automatic C++ code generation (using fsmc – Finite State Machine Compiler). This choice would reduce the risk for errors and generates an highly maintainable code.

2.2.2 AOS interaction

– The framework server-interface must be implemented using (linking) the *AOArbitratorLib* (by the AOS) and the *AOSLib* (by the AOArbitrator).

– ...

2.2.3 Wfs-Arbitrator and AdSec-Arbitrator interaction

– The framework client-interface must be implemented using (linking) the *AOArbitratorLib* (by the Wfs-Arbitrator and the AdSec-Arbitrator) and the *WfsArbitratorLibs and AdSecArbitratorLib* (by the MainArbitrator).

– ...

3 References

[1] Fini L., *AOS Functional Description*, LBT-AdOpt Technical Report No. LBT-AdOptSW.005, version 2.2, 10 Oct 2007

[2]

Doc_info_start
Title: AO-Arbitrator: requirements analysis
Document Type: Technical Report
Source: INAF-Osservatorio Astrofisico di Arcetri
Issued by:
Date_of_Issue:
Revised by:
Date_of_Revision:
Checked by:
Date_of_Check:
Accepted by:
Date_of_Acceptance:
Released by:
Date_of_Release:
File Type:MS-WORD
Local Name:
Category: WRITE THE CAN CATEGORY HERE
Sub-Category: WRITE THE CAN SUB-CATEGORY HERE
Assembly:
Sub-Assembly:
Part Name:
CAN designation: WRITE THE CAN NUMBER HERE
Revision:WRITE THE REVISION HERE
Doc_info_end