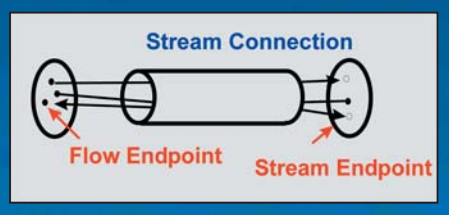
TRANSMITTING HUGE AMOUNTS OF DATA: DESIGN, IMPLEMENTATION AND PERFORMANCE OF THE BULK DATA TRANSFER MECHANISM IN ALMA ACS

P. Di Marcantonio[1], R. Cirami[1], B. Jeram[2], G. Chiozzi[2]

[1]INAF- Osservatorio Astronomico di Trieste, via G. B. Tiepolo 11, I-34131 Trieste, Italy
[2]European Southern Observatory, Karl-Schwarzschildstr. 2, D-85748 Garching, Germany

ALMA will be the largest millimetre wavelength astronomical interferometer in the world consisting of 64 12-meters antennas, and the need for transferring efficiently huge amounts of data arises consequently. For example, a typical output data rate expected from the correlator (the device responsible for the processing of raw digitized data from the antennas) will be of the order of 64 MB per second. Since all subsystems in ALMA rely on a communication infrastructure (ACS), which is CORBA-based, this poses some problems to meet the stringent QoS (quality-of-service) requirements. It is well known in fact that DOC (Distributed Object Computing) middleware, such as CORBA, increases the packet latency due to marshalling/de-marshalling, to usage of the IIOP protocol, etc. To cope with ALMA requirements and to overcome the CORBA potential bottleneck, we developed a transfer mechanism based on the *ACE/TAO CORBA Audio/Video (A/V) Streaming service*. This architecture uses CORBA for hand-shaking, but allows an efficient data transfer by creating out-of-bound stream(s) of data (i.e. bypassing the CORBA protocol), thus enabling ALMA applications to keep leveraging the inherent portability and flexibility benefits of the ACS middleware. Our infrastructure, which was put on the top of the ACE/TAO A/V Streaming service implementation, allows creating one or more *out-of-bound flows* in a simple way; each flow can be configured using different communication protocols (e.g. TCP, UDP) with a *measured efficiency comparable to that of a raw socket connection*.

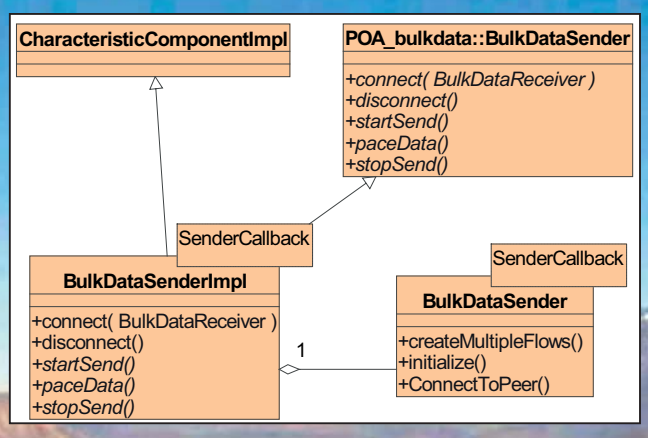
Basic terminology



The OMG CORBA A/V Streaming Services specification (on which the TAO A/V Streaming Service is based) defines a ***stream*** as a set of ***flows*** of data between objects, where a flow is a continuous sequence of frames in a clearly identified direction. A stream is terminated by a ***stream endpoint***, and can have multiple ***flow endpoints***, acting as a source or a sink of data.

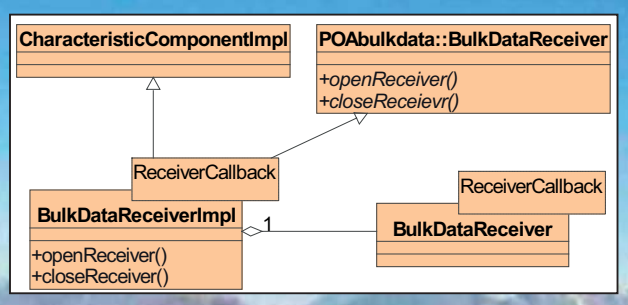
The ACS Bulk Data Transfer provides C++ classes and ACS Characteristic Components, which implement the features described. It allows to connect a ***Sender component*** (the producer of data) with a ***Receiver component*** (the consumer), creating dynamically as many flows as required, and provides also the necessary mechanism to mimic a multicast behaviour (a ***Distributer*** which connects multiple Receivers to one Sender).

The ACS Sender Component



The ACS Characteristic Component relative to the Sender is implemented as a C++ template class. The template parameter is a callback which can be used for sending asynchronous data. This callback provides methods for sending data at predetermined user-configurable time intervals. To allow sending data in a synchronous way, a default callback is provided, which disables the asynchronous mechanism.
As shown in the figure on the left, the BulkDataSenderImpl<T> template class realizes a component providing the implementation for the BulkDataSender IDL interface (represented in the diagram by the CORBA-generated POA_bulkdata::BulkDataSender skeleton class). BulkData-SenderImpl<T> provides a concrete implementation for the ***connect()*** and ***disconnect()*** methods using the C++ wrapper class (BulkDataSender<T>). The connect method is responsible for the connection establishment with a Receiver Component, passed as a parameter.

The ACS Receiver Component



TheACSCharacteristic Component relative to the Receiver is implemented also as a template class. The template parameter in this case is a callback, which has to be provided by the user and must be used to actually retrieve and manage the received parameters and data stream. The figure shows the class diagram for a Receiver Component. Two methods are implemented in this case: ***openReceiver()***, which reads e.g. from the Configuration Database all the connection parameters - as in the Sender case - and creates the required flow endpoints accordingly, and ***closeReceiver()***, used to close the connection.

The hand-shake mechanism

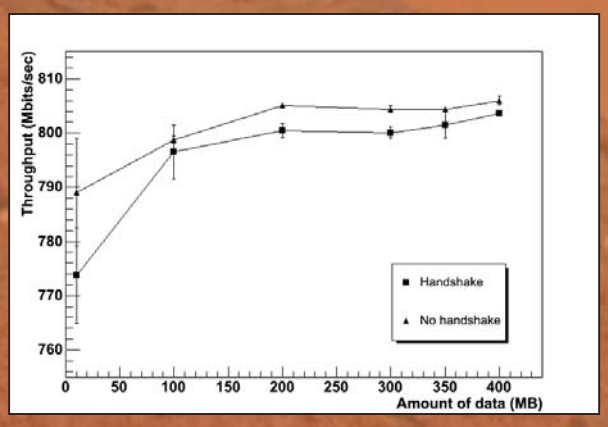
In the TAO A/V Streaming Service, the Sender/Receiver architecture is implemented by using the *ACE Reactor Pattern*, and uses a callback mechanism to actually manage the incoming data stream. The provided TAO A/V Callback class offers methods to fulfil this purpose, but has the following limitations:

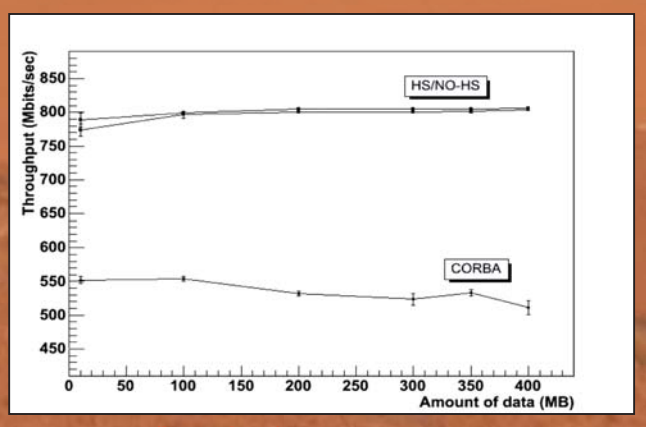
1. there is no possibility to send short parameters directly when a start is issued (for example an UID to characterize the forthcoming frame, a string containing a filename to be opened, etc.);
2. a synchronization problem occurs.

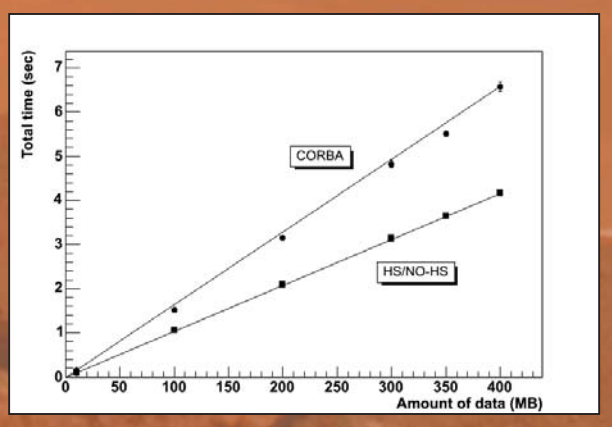
Point 2 is quite subtle. Data sent by the Sender are first received in the TCP-receive memory buffer of the involved host (whose typical default size for Linux Red Hat 9.0 is around 85 KB). Being the ACE_reactor event-driven, as soon as data are available, the pre-registered callback method is called and data are consumed. The limitation is that internally the TAO A/V reads data only in chunks of 8192 bytes. It could happen therefore that the Sender receives the acknowledgement of the last frame received even if the data are still not fully consumed on the Receiver side (they are actually stored in the host TCP receive buffer, but are not read yet). In this case a stop could be issued to early spoiling the last part of the received stream.
In order to overcome this problem, we implemented a ***hand-shake protocol*** on top of this architecture. Before sending the raw data, a control frame is sent and analyzed. The control frame contains the information (an ID) on whether the forthcoming stream is a parameter or the bulk of data, and the number of expected bytes length. The ID allows to distinguish between parameters and data, whereas the bytes length information permits to manage and overcome the synchronization problem.

Achieved Performance

In order to evaluate the performance of the ACS Bulk Data Transfer we developed and implemented two ACS Components. To obtain meaningful results we deployed the two components on two Compaq PCs (P4, 3.0 GHz) equipped with 1GB RAM and 80 GB HD connected via a ***1Gbit Ethernet network***. Both PCs were isolated from the Institute LAN to avoid external network loads. Linux Red Hat 9.0 operating system and ACS 4.1.2 were installed on both machines.







The figures above represent the results of our performance tests. The *throughput*, i.e., the number of bits per second, vs different amounts of transmitted data is shown on the left and middle pictures. ***It clearly appears that the ACS Bulk Data mechanism is always better than using simple CORBA call (the estimated gain is around 30%) and that the hand-shake mechanism does not introduce relevant performance penalties respect to the no hand-shake case.***
The right picture shows and compares the linearity in the three cases. The hand-shake and no hand-shake samples are overlapped proving that also in the case of the hand-shake protocol the linearity is preserved.