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*

ABSTRACT

We present and discuss the design, implementation and performance of the bulk data transfer mechanism developed in the framework of the ALMA (Atacama Large Millimeter Array) Common Software (ACS) [2][6]. 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 [1]. 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 handshaking, 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 (a flow is a continuous sequence of frames in a clearly identified direction); 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.

We designed and implemented also a *Distributer* model, which mimics a multicast behaviour. One or more receivers can subscribe to a common object (the *Distributer*) which receives data from one sender (e.g. the correlator), and dispatches them to all the subscribed receivers using out-of-bound connections.

INTRODUCTION

The whole software infrastructure for ALMA is based on ACS (ALMA Common Software) (for a detailed description see [2][6]), which is a set of application frameworks built on top of CORBA. This poses some problems to meet the stringent QoS requirements for data transfer. In order to overcome this bottleneck, we have implemented a transfer mechanism based on the ACE/TAO CORBA Audio/Video (A/V) Streaming service [3], the ***ACS Bulk Data Transfer***. This mechanism uses an out-of-bound connection for the data stream (adopting communication protocols like TCP), thus bypassing the CORBA protocol and, at the same time, using CORBA for handshaking and leveraging the benefits of ACS middleware.

This paper outlines the design and implementation issues of the ACS Bulk Data Transfer and analyzes the achieved performances. We start by introducing some new terminology, and in the subsequent sections the provided tool is described.

The OMG CORBA A/V Streaming Services specification [4] (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 as a sink of data (see Figure 1).

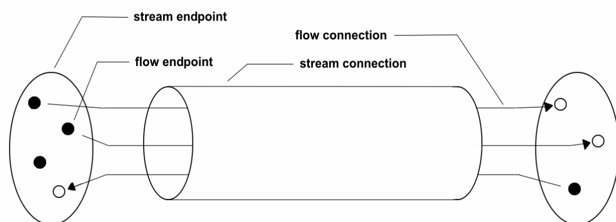

Figure 1: Basic stream configuration

The ACS Bulk Data Transfer provides C++ classes and ACS Characteristic Components, which implement the features described above (we assume that typical ACS concepts/paradigms like Components, Manager, Containers, etc. are familiar to the reader, otherwise see for example [2][6]). 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). Since the multicast is available only with the UDP protocol which does not guarantee the delivery of all the packets, the necessity to mimic the multicast behaviour arises.

DESIGN AND IMPLEMENTATION

The ACS Bulk Data Transfer provides a wrapper and an adaptation of the CORBA A/V Streaming Service (in the TAO implementation) to ACS, hiding most of its complexity from the user. C++ classes have been created (at present only C++ implementation is provided), which allow to create a stream (and thus a connection between the Sender and the Receiver) adding to it as many flows as needed. Once the connection has been successfully established, the Sender can immediately start to send data either in a synchronous or in an asynchronous way. The Receiver, on the other side, can receive data only in an asynchronous way by using a callback mechanism.

Besides C++ classes, an ACS Characteristic Component has been implemented, which contains and uses these C++ classes, offering the developer user-friendly IDL programming interfaces. They are briefly described in the following two subsections.

Sender ACS 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 class provides methods for sending data at predetermined user-configurable time intervals. To allow sending data in a synchronous way, a default callback class is provided, which disables the asynchronous mechanism.

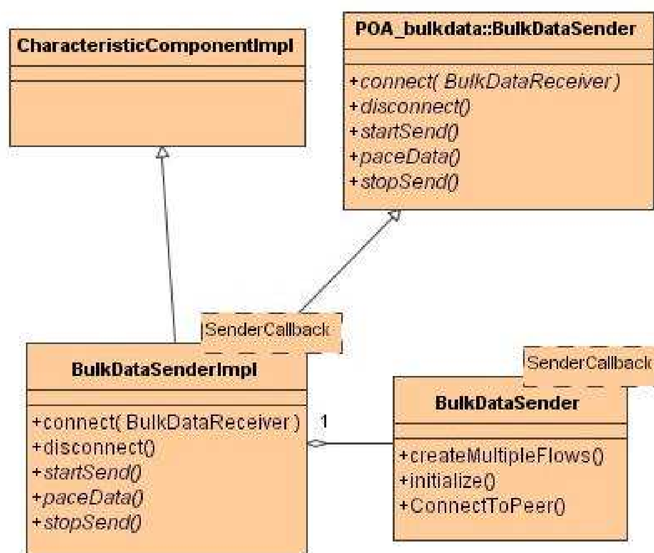

Figure 2: ACS Sender Component class diagram

As shown in Figure 2, 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). BulkDataSenderImpl<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. By reading from the Configuration Database (see [2]) the connection parameters such as the number of flows of the stream, the protocol (TCP or

UDP), and the host and port number, the *connect()* method fully manages the creation of appropriate flow endpoints with different settings. The other three methods (*startSend(),paceData(),stopSend()*) are purely abstract and must be implemented by the user. Once the out-of-bound connection is correctly established, they are used to actually send short parameters (*startSend()*) and huge amounts of data (*paceData()*).

Receiver ACS Component

The ACS Characteristic Component relative to the Receiver is implemented also as a template class. The template parameter in this case is a callback class, which has to be provided by the user and must be used to actually retrieve and manage the received parameters and data stream (see description in the next section).

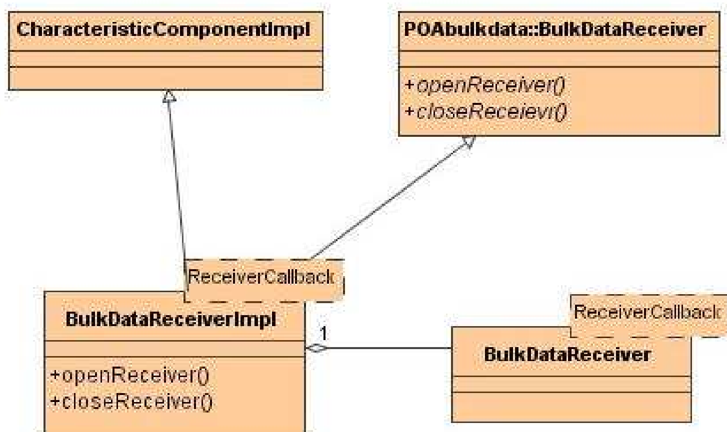

Figure 3: ACS Receiver Component class diagram

Figure 3 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.

As in the Sender case, the actual implementation is delegated to the C++ class BulkDataReceiver<T>.

Such a design and implementation proves to be very flexible: the number of flows can be different on either part, but only those that match against some criteria (like protocol, direction, and name according to A/V rules etc.) are then actually connected. Note that the managing of flow creation, connection establishing, reading of parameters etc., is completely hidden from the user point of view. By providing the required callback to manage the received stream and by providing the data to be sent, an interested user can use directly the Sender and Receiver Components without the need for further development.

Receiver Callback and High-Level Hand-shake mechanism

In the TAO A/V Streaming Service, the Sender/Receiver architecture is implemented by using the ACE Reactor Pattern (see [5]), and uses a callback mechanism to actually manage the incoming data stream. The provided *TAO_AV_Callback* class offers three methods to fulfil this purpose: *handle_start()* and *handle_stop()*, which react when a start/stop is issued on a specific flow, and a *receive_frame (ACE_Message_block *frame)*, which is used to get the received data, 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 reactor concrete event handler is the *receive_frame()* method, as described before). 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.

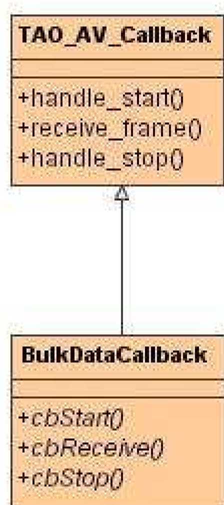

Figure 4: ACS Callback class

In order to overcome this problem, we implemented a hand-shake protocol on top of this architecture, by inheriting from the TAO_AV_Callback (as shown in Figure 4), and adding internally a new state management (not shown). Before sending the raw data, a control frame is sent and analyzed by the *BulkDataCallback* callback class. 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 call internally the appropriate methods (*cbStart(ACE_Message_Block * param)/cbRecei-ve(ACE_Message_Block * frame)*) to distinguish between parameters and data, whereas the bytes length information permits to manage and overcome the synchronization problem.

The hand-shake mechanism described above is completely hidden to the user. To receive fully synchronized parameters/data she/he must only inherit from BulkDataCallback and implement the three abstract methods (see Figure 4), without knowing anything about what happens below. Of course this causes some performance penalties, which is the topic of the next section.

ACHIEVED PERFORMANCE

In order to evaluate the performance of the ACS Bulk Data Transfer we developed and implemented two ACS Components (hereafter simply called the Sender and the Receiver), following the design described in the previous sections. The aim of the experiment was twofold:

- measure the throughput, i.e., the number of bits per second, sending data of different size from a Sender to a Receiver;
- compare the measured throughput using three different mechanisms:
  - simple CORBA call, i.e. sending an array of chars as a parameter of a CORBA method;
  - ACS A/V with the hand-shake mechanism (hereafter called HS);
  - ACS A/V without the hand-shake mechanism (hereafter called NO-HS).

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 results are depicted in Figure 5, where on the X axis the buffer sizes sent from a Sender to the Receiver are reported, and on the Y axis the measured throughput. Every point is an average of several samples. The error bars represent the error on the mean. The figure shows that:

- the throughput obtained via the Bulk Data mechanism either with or without the hand-shake mechanism is always better than using simple CORBA call. The estimated gain is about 30%;
- the hand-shake mechanism introduces some performance penalties (expected and discussed below, see Figure 6), but only of the order of 0.5%;
- the CORBA performance is always worse than the A/V streaming, and shows a fall for increasing buffer sizes.

Figure 6 shows the comparison between data received either with or without the hand-shake mechanism. The performance penalties introduced by the hand-shake mechanism is largely due to the need to overcome the synchronization problem.

An incoming *Stop* call is blocked until all the received data are correctly consumed. Besides this, in order to correctly distinguish between an incoming parameter and a datum, we are forced to call the underlying TAO A/V CORBA objects. Note however that these CORBA calls do not pass any value and they are used just for synchronization purposes. Stream data are always sent with an out-of-bound connection. Therefore, as shown in Figure 6, the overall overhead introduced by the hand-shake mechanism is limited and comparable with the no hand-shake protocol.

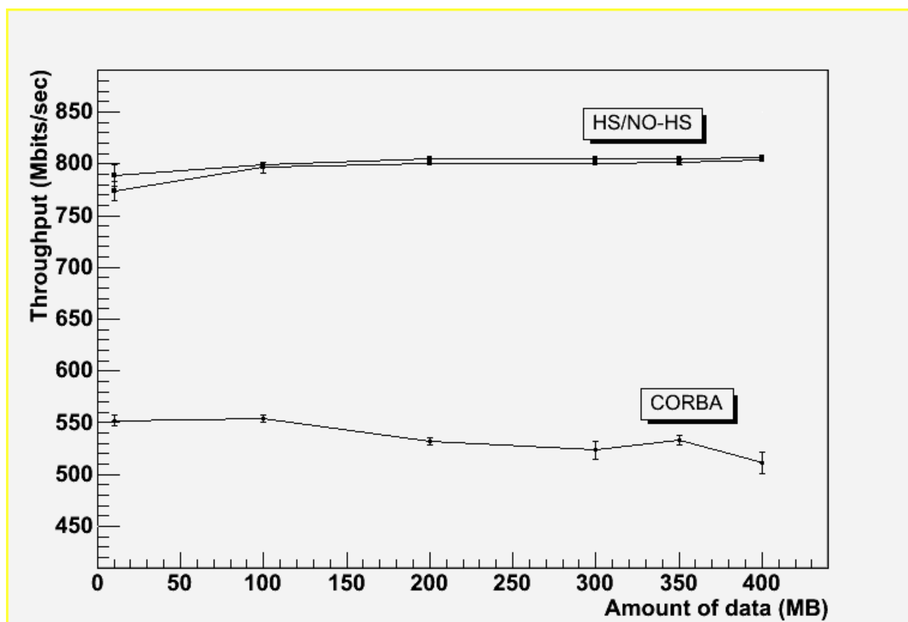

Figure 5: Overall measured throughput performances when transmitting various amounts of data.

Finally, Figure 7 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.

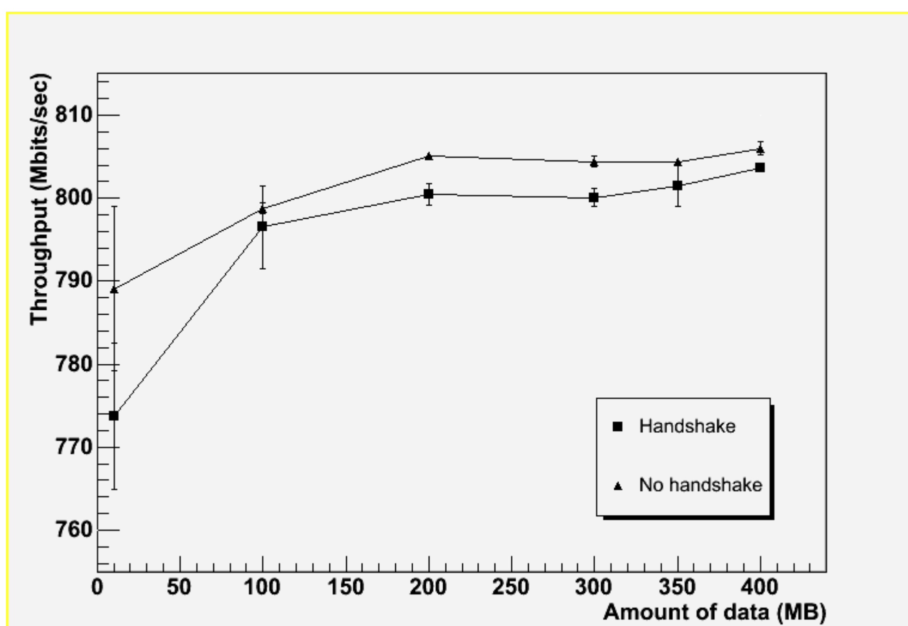

Figure 6: Performance comparison between hand-shake and no hand-shake mechanism; note that with increasing amount of data, the overhead due to the hand-shake mechanism decreases.

DISTRIBUTER

Besides the described point-to-point communication model between a Sender Component and a Receiver, we are currently implementing also a *Distributer* model, which mimics a multicast behaviour. The basic idea is that different Receivers, willing to get data from the same Sender, connect to a *Distributer* Component. This *Distributer* receives data from the Sender and manages the data dispatching by using out-of-bound connections. Such a design is necessary since the TCP

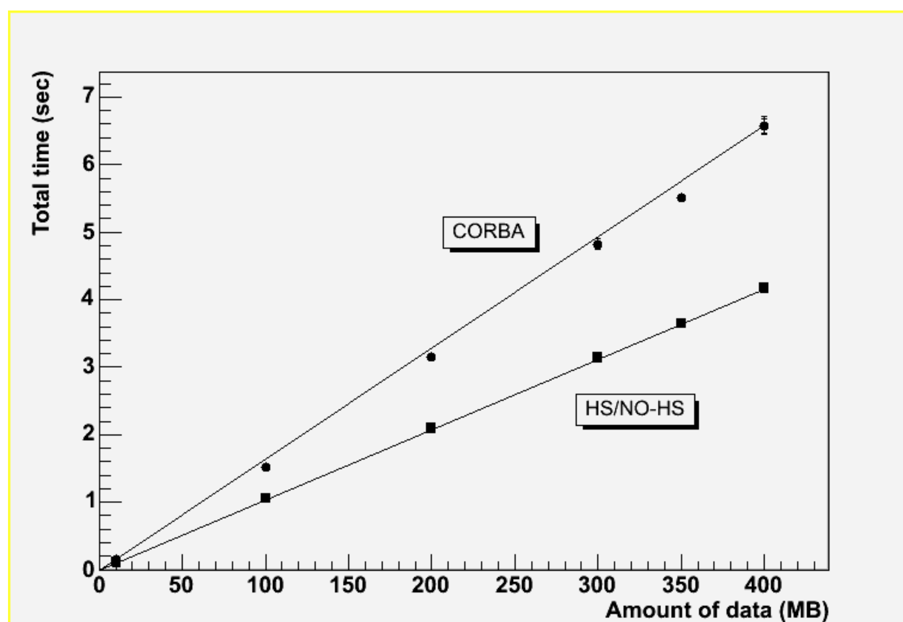

Figure 7: Throughput linearity in the three cases. The straight lines are just superimposed to better show the achieved linearity.

protocol does not support multicasting and we must guarantee that all data are correctly delivered. Since the *Distributer* can be placed on a different host than the Sender, we can shield the Sender itself (often a critical component in the system) from changing load due to a different number of Receivers. Only the *Distributer* will be affect by such problems. Anyway, the *Distributer* model is still under implementation and will not be described further in this paper.

CONCLUSIONS

The paper describes the design and the implementation of the ACS Bulk Data Transfer mechanism. Performance tests at our Institute clearly show that it improves the overall performances when transferring huge amount of data, comparing to a simple CORBA call. The final gain could be estimated in the order of 30%. Work is ongoing to implement also a Distributer model to mimic multicast behaviour.

REFERENCES

[1] J. Pisano et al., “ALMA correlator computer system”, Proceedings of SPIE vol. 5496, Glasgow 2004, 146.
[2] G. Chiozzi et al., “The ALMA common software: a developer friendly CORBA-based framework”, Proceedings of SPIE vol. 5496, Glasgow 2004, 205.
[3] N. Surendran et al., “The Design and Performance of a CORBA Audio/Video Streaming Service”, Proceedings of HICSS-32 vol. 8, Hawaii 1999, 8043.
[4] OMG Audio/Video Streams Specification, v.1.0, http://www.omg.org/cgi-bin/doc?formal/2000-01-03.
[5] D. C. Schmidt, S. D. Huston, “C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks”, Addison Wesley, 2002.
[6] G.Chiozzi et al, “The ALMA Common Software (ACS): status and developments”, ICALEPCS’2005, Geneva, Switzerland, October 2005.