Bulk data transfer distributer: a high performance multicast model in ALMA ACS

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

[1]INAF-OAT, 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

A high performance multicast model for the bulk data transfer mechanism in the ALMA (Atacama Large Millimeter Array) Common Software (ACS) is presented. The ALMA astronomical interferometer will consist of at least 50 12-m antennas operating at millimeter wavelength. The whole software infrastructure for ALMA is based on ACS, which is a set of application frameworks built on top of CORBA. To cope with the very strong requirements for the amount of data that needs to be transported by the software communication channels of the ALMA subsystems (a typical output data rate expected from the Correlator is of the order of 64 MB per second) and with the potential CORBA bottleneck due to parameter marshalling/de-marshalling, usage of IIOP protocol, etc., a transfer mechanism based on the ACE/TAO CORBA Audio/Video (A/V) Streaming Service has been developed. The ACS Bulk Data Transfer architecture bypasses the CORBA protocol with an on out-of-bound connection for the data streams (transmitting data directly in TCP or UDP format), using at the same time CORBA for handshaking and leveraging the benefits of ACS middleware. Such a mechanism has proven to be capable of high performances, of the order of 800 Mbits per second on a 1Gbit Ethernet network.
Besides a point-to-point communication model, the ACS Bulk Data Transfer provides a multicast model. Since the TCP protocol does not support multicasting and all the data must be correctly delivered to all ALMA subsystems, a distributer mechanism has been developed. This paper focuses on the ACS Bulk Data Distributer, which mimics a multicast behaviour managing data dispatching to all receivers willing to get data from the same sender.

1. INTRODUCTION

The ALMA Common Software (ACS) is a set of application frameworks built on top of CORBA to provide a common software infrastructure to all partners in the ALMA collaboration (for a detailed description see [1][2]). Since ALMA will be the largest astronomical interferometer in the world, consisting of at least 50 12-m antennas operating at millimeter wavelength, the need for transferring efficiently huge amounts of data arise consequently. For example, a typical output data rate expected from the Correlator (the device responsible for the processing of raw digitalized data from the antennas) will be of the order of 64 MB per second [3].
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 [4], the ACS Bulk Data Transfer. 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. We developed in this way a transfer mechanism capable of high performances, of the order of 800 Mbits per second on a 1Gbit Ethernet network [5].
Besides a point-to-point communication model, which allows the data transfer between one sender and one receiver, the ACS Bulk Data Transfer provides a multicast model. Since the TCP protocol does not support multicasting and all the data must be correctly delivered to all ALMA subsystems, a distributer mechanism has been developed. It is possible in this way to transfer data from one sender to more connected receivers.
This paper focuses on the ACS Bulk Data Distributer, 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. Locating the Distributer on a different computer has also the advantage of reducing the network/CPU load.
Before proceeding with the design description and implementation of the ACS Bulk Data Distributer, it is necessary to introduce some terminology (see Fig. 1).
The CORBA A/V Streaming Service specification [6] defines a flow as a continuous sequence of frames in a clearly identified direction between (two) multimedia devices. A stream is defined as a set of flows between two objects, and is terminated by a stream endpoint. A stream endpoint can have multiple flow endpoints, acting as a source or as a sink of data.

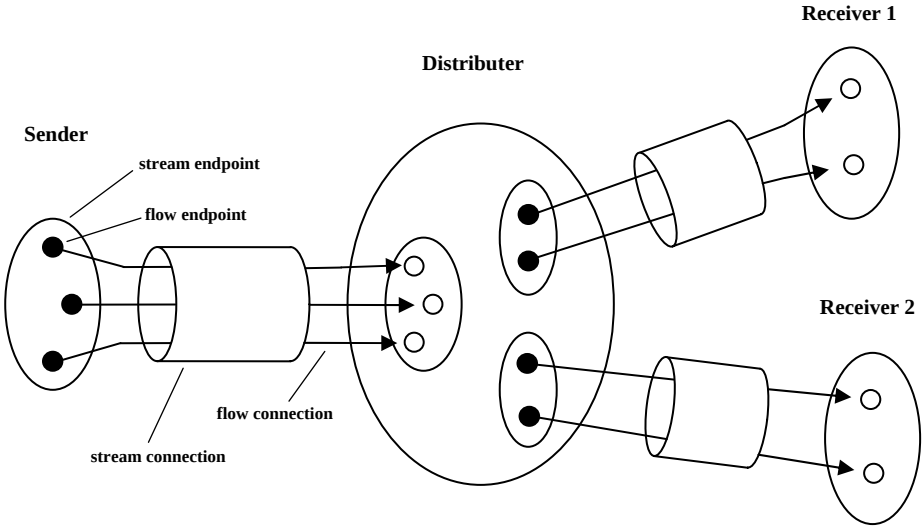

Fig. 1. Stream configuration for Sender, Distributer and Receivers (Black spots: sender flow endpoints. White spots: receiver flow endpoints)

The ACS Bulk Data Transfer provides C++ classes and ACS Characteristic Components which implement the features described above (for ACS see [1][2]). In the peer-to-peer communication model, it allows to connect a sender component (the producer of data) with a receiver component (the consumer), creating dynamically as many flows as required. In the multicast model a sender component send data to the distributer component which, in turn, delivers the data to one or more connected receiver components.

2. 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 for the sender, distributer and receiver components (at present only C++ implementation is provided). These classes allow to create a stream between the ACS Bulk Data components adding to it as many flows as needed.
Once the connection has been successfully established, the Sender can immediately start to send data to the Distributer in a synchronous (blocking) way.
The Distributer and the connected Receivers, on the other side, can receive data only in an asynchronous way, by means of a callback called automatically by the ACE Reactor implemented in the TAO A/V. For each distributer or receiver flow the ACE Reactor, as soon as data are available, calls the associated callback. As soon as the Distributer receives the data inside its callback, it forwards them immediately to all connected Receivers, mimicking in this way a multicast behaviour.

Besides C++ classes, ACS Characteristic Components for the Sender, Receiver and Distributer have been implemented, which contains and uses the aforementioned C++ classes, offering the developer user-friendly IDL (CORBA Interface Definition Language) programming interfaces. These Components are briefly described in the following subsections.

2.1. 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. For the Distributer such asynchronous mechanism has not been implemented yet, and only the synchronous mechanism is provided.
As shown in Fig. 2, the *BulkDataSenderImpl<TSenderCallback>* 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<TSenderCallback>* provides a concrete implementation for the *connect* and *disconnect* methods using the contained C++ wrapper class (*BulkDataSender<TSenderCallback>*). The *connect* method is responsible for the connection establishment with the distributer component, passed as parameter. By reading from the Configuration Database [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* and *stopSend*) are purely abstract and must be implemented by the user. Once the out-of-bound connection is correctly established, these methods are used to actually send short parameters (*startSend*), huge amounts of data (*paceData*) and to terminate the data transfer (*stopSend*).

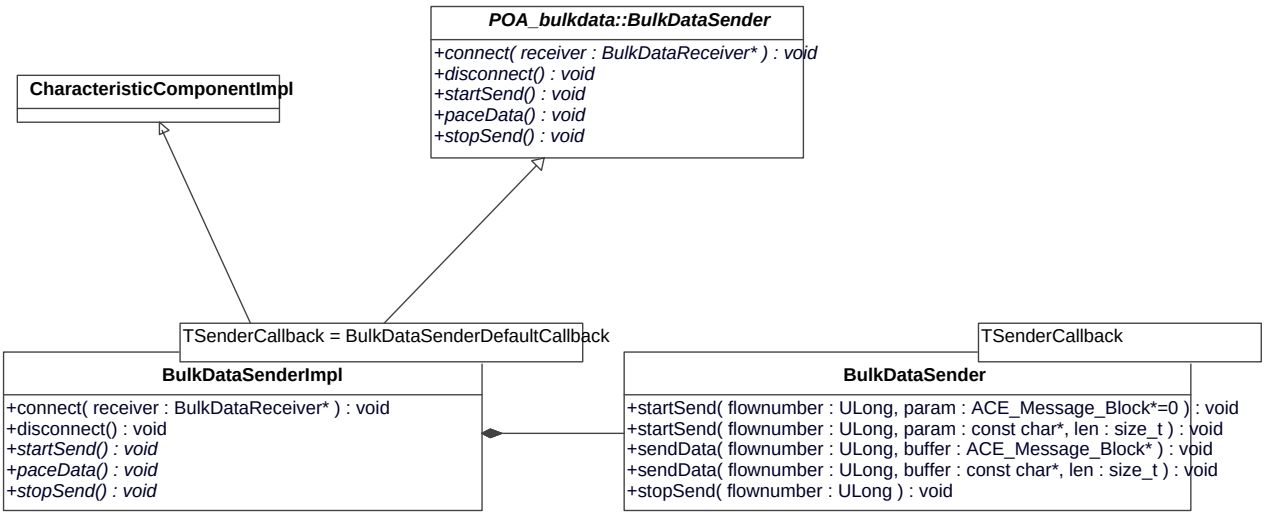

Fig. 2: Sender ACS component class diagram

2.2. 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 2.2.1).

Fig. 3 shows the class diagram for a receiver component. Two methods are implemented in this case: *openReceiver*, which reads 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.

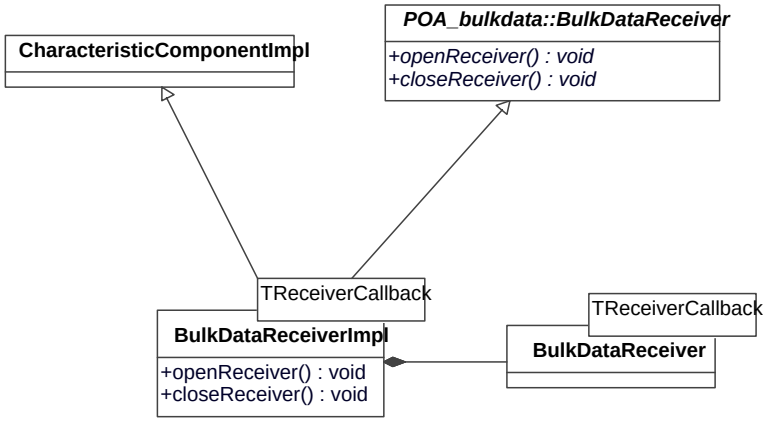

Fig. 3: Receiver ACS component class diagram

As in the sender case, the actual implementation is delegated to the C++ class *BulkDataReceiver<TReceiverCallback>*. 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.

2.2.1. Receiver Callback

The responsibility of the receiver callback is to receive parameters and data from the Distributer. This callback mechanism derives directly from the TAO A/V Streaming Service, where the Sender/Receiver architecture is implemented by using the ACE Reactor Pattern [7], which uses a callback mechanism to actually manage the incoming data stream.
The provided *TAO_AV_Callback* class offers three methods to fulfill this purpose (see Fig. 4): *handle_start* and *handle_stop*, which react when a start/stop is issued on a specific flow (CORBA calls), and a *receive_frame*, which is used to get the received data and is triggered by the ACE Reactor. This mechanism 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 Distributer are first received in the TCP-receive memory buffer of the involved host (whose typical default size for Scientific Linux 4.1 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

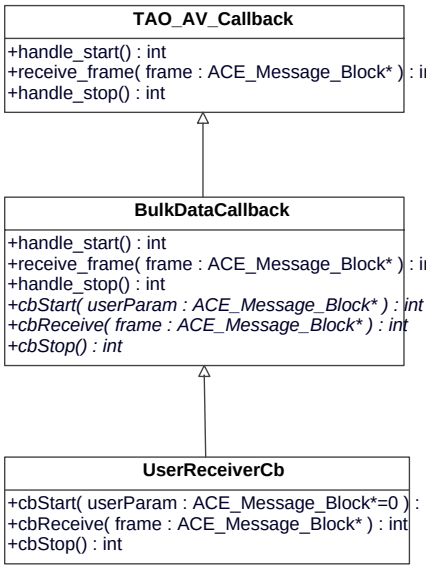

Fig. 4: Receiver callback class

is that internally the TAO A/V reads data only in chunks of 8192 bytes. It could happen therefore that the Distributer 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, by inheriting from the *TAO_AV_Callback* (as shown in Fig. 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/cbReceive*) to distinguish between parameters and data, whereas the bytes length information permits to manage and overcome the synchronization problem.
This hand-shake mechanism allows the user to receive fully synchronized data. What the user has to do is only to inherit from *BulkDataCallback* and implement the three abstract methods (*cbStart*, *cbReceive* and *cbStop*), without knowing anything about what happens below.

2.3. Distributer ACS Component

The ACS Bulk Data Distributer acts as a Receiver towards the Sender and as a Sender towards the Receivers willing to get the dispatched data, so the ACS Characteristic Component relative to the Distributer inherits form both the sender and receiver interfaces (see Fig. 5). The *BulkDataReceiverDistr* interface is used for internal purposes. It is implemented as a template class with two template parameters: the sender callback, which takes the default value *BulkDataSenderDefaultCallback* (see 2.1), and the distributer receiver callback (*BulkDataDistributerCb*), used to receive the data from the Sender (see Fig. 6). This callback manages and dispatches the received data to all registered Receivers, taking care of the possible receiver timeout and availability (see 2.3.1).

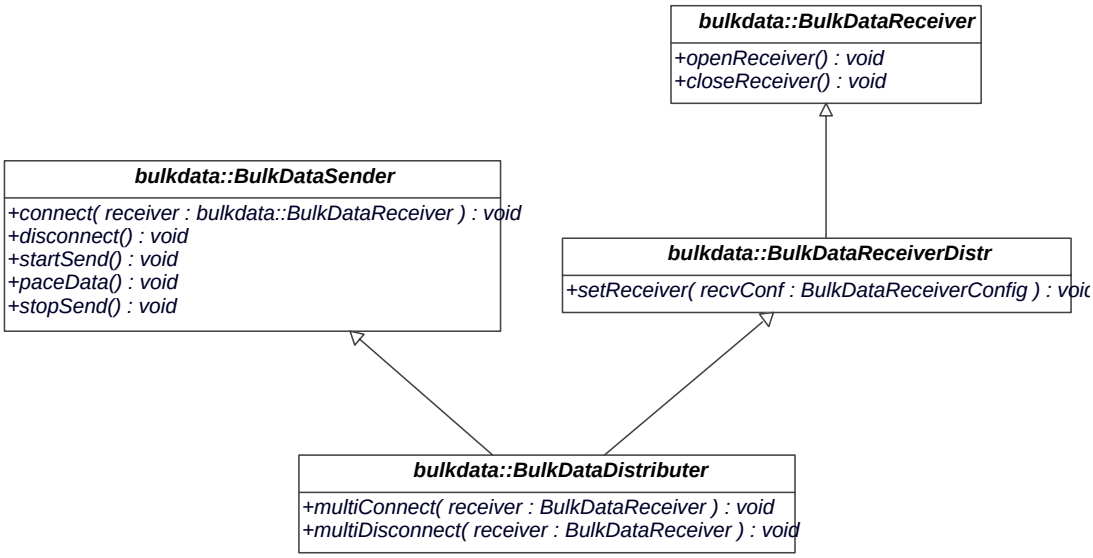

Fig. 5: Class diagram for Distributer idl interfaces

The Sender connects to the Distributer in the usual way, with the *connect* method, passing the distributer component as parameter. The distributer *multiconnect* method allows the Receivers to register themselves into the Distributer. It delegates to the *multiconnect* method of the contained *BulkDataDistributer* object the task of creating the connection with the Receivers. Inside the Distributer, for each connecting Receiver an object of the class *BulkDataSender* is istantiated. Then the connection is created between this new Sender object and the Receiver, with the number of flows

specified inside the Configuration Database. A hash table is maintained internally, which associates each Receiver with the corresponding sender object.

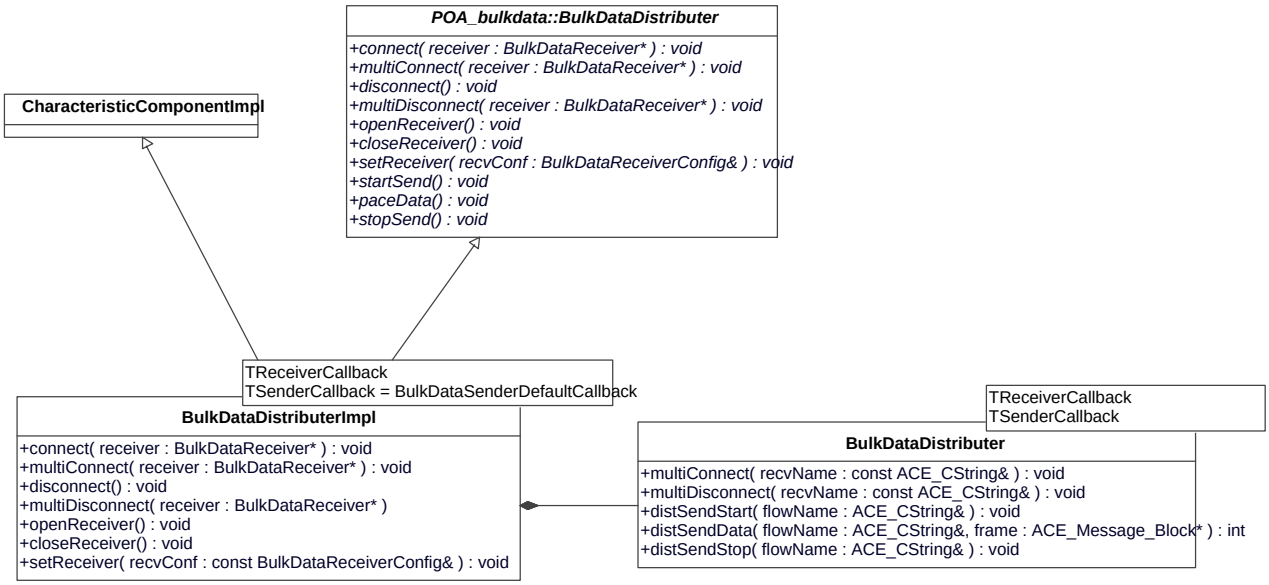

Fig. 6: ACS distributer component class diagram

When the Distributer receives parameters or data from the Sender, its receiving callback (*BulkDataDistributerCb*) delegates to the contained distributer class the task of forwarding the incoming parameters and data to all the connected Receivers. At present this forwarding mechanism is implemented in a sequential way, i. e. the internal hash table which holds the information of the connected Receivers is read sequentially. This is a known bottleneck (see 3), and will be improved in the next ACS Bulk Data releases.

2.3.1. Distributer Callback and High-Level Hand-shake mechanism

The responsibility of the distributer callback is to receive data from the Sender (from one or more flows) and deliver them correctly to all the connected Receivers.
Also in this case we have implemented a callback (*BulkDataDistributerCb*) class which derives from *TAO_AV_Callback* and which implements the hand-shake protocol as in the receiver callback case (see Fig. 7).
As soon as the distributer callback receives parameter and/or data from the sender, it must deliver them immediately to all connected Receivers, delegating this task to the contained distributer class (see 2.3). In order to do this correctly, it must take into account the fact the one or more Receivers may not be ready to receive the data (some receiver flows may take too long elaborating the data or some Receivers may not be available). Therefore we have implemented an appropriate mechanism for addressing this problem. The distributer callback contains an internal table that holds the information of the status of all the flows of all connected Receivers. In case of error, the receiver status is checked inside the table and, if it is not available, the data is not sent to the Receiver on that particular flow. At the same

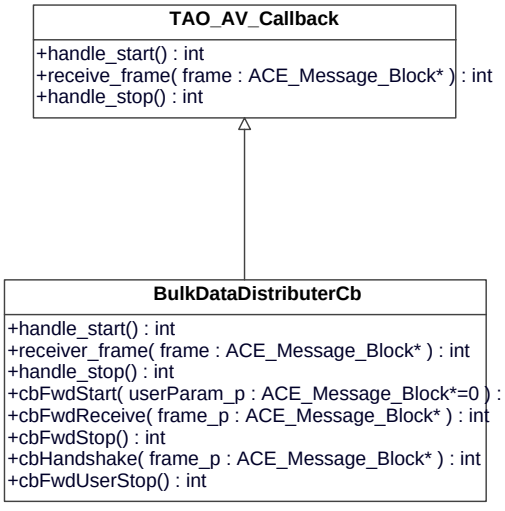

Fig. 7: Distributer callback class

time, this mechanism allows to inform the Sender that a particular receiver is in a timeout state or is not working properly, so that it can take the appropriate action.

3. BULK DATA DISTRIBUTER PERFORMANCES

In order to evaluate the performance of the ACS Bulk Data Distributer we developed and implemented three ACS Components (hereafter simply called the Sender, the Distributer and the Receiver), following the design described in the previous sections. The aim of the experiment was to measure the throughput, i.e., the number of bits per second, sending data of different size from a Sender to one or more Receivers passing through a Distributer. In all the performance tests the data were sent on one flow.
For the measurements we used 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, but to be more realistic the two PC were connected using two Gbit switches. Scientific Linux 4.1 operating system and ACS 5.0.2 were installed on both machines.
The results are shown in Fig. 8 and Fig. 9. On the X axis the buffer sizes sent from a Sender to the Receiver(s) are reported, and on the Y axis the measured throughput. Every point is an average of 100 samples. The error bars represent the error on the mean.
In all the tests, the Sender and the Distributer were located on one PC, whereas the Receiver(s) on the other one.
Fig. 8 shows that:

- The throughput between a Sender and a single Receiver is more than 800 Mbits/sec, confirming the results obtained in [5], but using now a different operating system and ASC release.
- The Distributer introduces a penalty of performances of the order of 300 Mbits/sec. This is an expected result, since the Distributer introduces a further data processing step in the data flow. Data must be received, read and forwarded to the Receiver. The Distributer then has to wait until the Receiver has fully consumed the data in order to be synchronized. This could be considered the best performance achievable fulfilling the synchronization requirement.
- The measured throughput in the range of the sent buffer size (10-400 MB) is nearly constant.

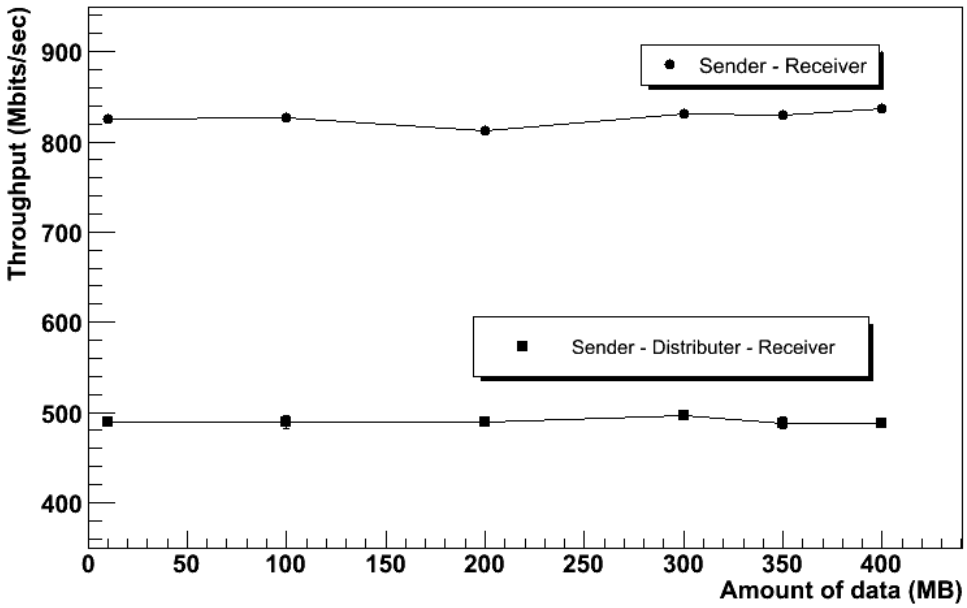

Fig. 8: Throughput performances measured with Sender-Receiver and Sender-Distributer-Receiver configurations

Fig. 9 shows the throughput when one or more Receivers are connected to the Distributer. It can be seen that for every connected Receiver the performance penalty increases and the overall throughput for four connected Receivers is of the order of 180 Mbits/sec. As described in 2.3, this is due to the sequential way of sending the data. This mechanism will be improved in the next Bulk Data releases using e.g. a thread pool or the Half-Sync/Half-Async pattern [7].

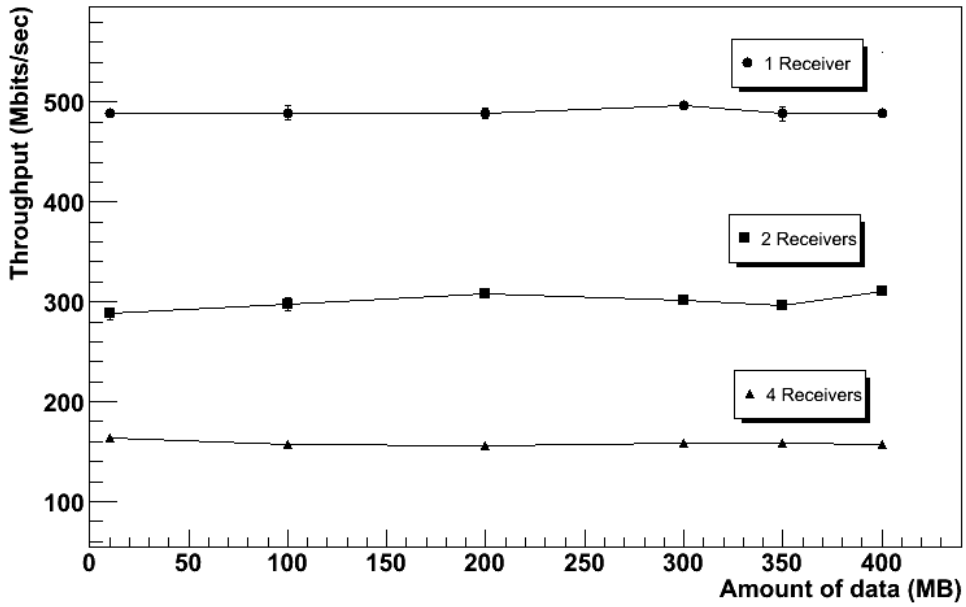

Fig. 9: Throughput performances measured with one, two and four Receivers connected to the Distributer

4. CONCLUSION

This paper describes the design and the implementation of the ACS Bulk Data Distributer. Performance tests at our Institute clearly shows that introducing a distributer model in the ACS Bulk Data Transfer mechanism introduces an expected performance penalty, estimated of the order of about 40%. If more than one Receiver is connected to the Distributer the performances obviously decrease. This could be improved by using a different way of forwarding data and will be analyzed/implemented in the next ACS Bulk Data Transfer releases.

REFERENCES

1. G. Chiozzi et al., “Application development using the ALMA common software”, these proceedings.
2. G. Chiozzi et al., “The ALMA common software: a developer friendly CORBA-based framework”, Proceedings of SPIE vol. 5496, Glasgow 2004, 205.
3. J. Pisano et al., “ALMA correlator computer system”, Proceedings of SPIE vol. 5496, Glasgow 2004, 146.
4. N. Surendran et al., “The Design and Performance of a CORBA Audio/Video Streaming Service”, Proceedings of HICSS-32 vol. 8, Hawaii 1999, 8043.
5. P. Di Marcantonio, R. Cirami, B. Jeram, G. Chiozzi, “Transmitting huge amounts of data: design, implementation and performance of the Bulk Data Transfer mechanism in ALMA ACS”, ICALEPCS 2005, Geneva, Switzerland, October 2005.
6. OMG Audio/Video Streams Specification, v.1.0, http://www.omg.org/cgi-bin/doc?formal/2000-01-03.
7. D. C. Schmidt, S. D. Huston, “C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks”, Addison Wesley, 2002.