Experiences With Communication Services for Advanced Network Applications

John Bresnahan, Ian Foster, Joseph Insley, Brian Toonen, and Steven Tuecke
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL, U.S.A.

Abstract *Type the abstract (100 to 150 words) using Italic font with point size 10. The abstract is an essential part of the paper. Use short, direct, and complete sentences. It should be as brief as possible and concise. [You are requested to also email a copy of your abstract (100 to 150 words) in ASCII (plain text) to hra@cs.uga.edu together with the title and list of authors of your paper. These abstracts may be published in a Book of Abstracts.]*

Keywords: a maximum of six keywords

1 Introduction

The Computed MicroTomography (CMT) application [Ref: XXX] is an example of an emerging class of advanced networking, or grid, applications. In this application, a large number of 2D images of objects are captured using the Advanced Photon Source (APS), a bright x-ray source at Argonne National Laboratory. As the images are captured, they are transferred over high speed network to a parallel supercomputer, which performs an incremental 3D reconstruction of the material densities from the 2D slices. The 3D dataset is then incrementally transferred over another high speed network to a collaborative visualization system.

The visualization system uses SGI's Volumizer library to produce high-quality, 3D volume rendered images of the dataset. This dataset is displayed and controlled in multiple ways:

- **Virtual reality display:** A CAVE or Immersadesk provides a high-quality, 3D stereo display of the dataset. The user can control the display through the use of a virtual control panel within the virtual environment. This control panel allows operations such as rotation, cropping of the volume, and assigning color and opacity to dataset voxels.

- **Desktop display:** Rather than rewriting the volume renderer to run on the less capable hardware that is available in a desktop, we instead chose to use remotely rendered video for the desktop display. The same SGI-based volume rendering hardware and software is used to produce high-quality volume rendered images, but instead of displaying them directly on an Idesk, the frames are instead captured, compressed, and sent over the network for display using stardard network video display tools such as vic. A Java control panel runs locally on the desktop, and is used to control the remote rendering in the same manner as the Idesk control panel. Multiple desktops can simultaneously view the video, and control the volume rendering from the Java control panel.

- **Web display:** The same software can also capture individual images from the scene and put them on a web page, for a very low-end solution that provides high quality images.

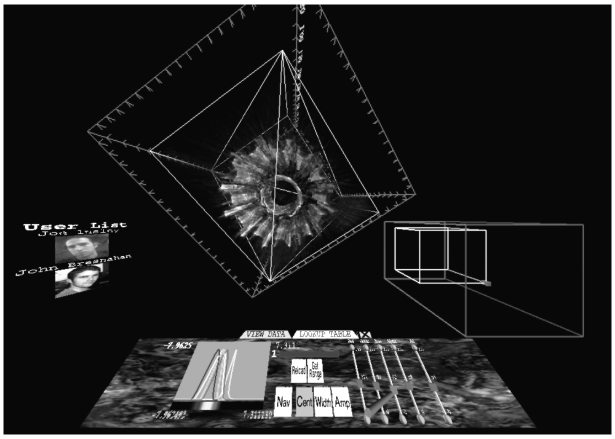

Figure 1: *A screen shot of the ImmersaDesk taken during a collaborative session with two users.*

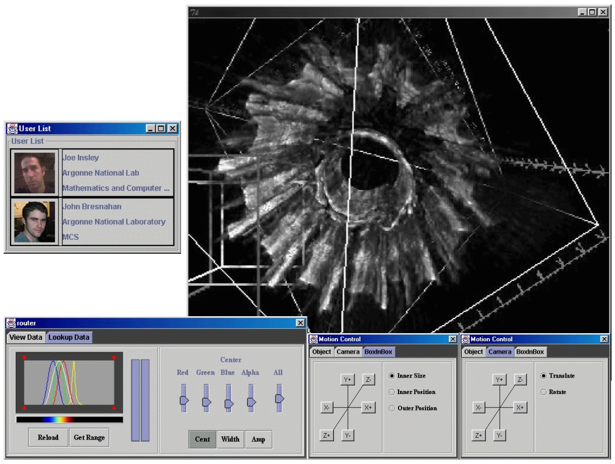

Figure 2: *A screen shot of a low-resolution graphics workstation taken during a collaborative session with two users.*

The virtual reality and desktop displays are fully linked, allowing collaborative viewing of datasets even between these very different display devices. All operations on the Idesk are reflected on the Java-based desktop control panel, and vice versa.

2 Communication Requirements

This application requires the simultaneous use of many underlying communication structures:

- The transfer of 2D images from the APS to the supercomputer, and of 3D datasets from the supercomputer to the visualization system, requires high-bandwidth, unicast communication.
- Communication within the parallel 3D reconstruction program requires very high bandwidth and low latency communication, as is typically available on parallel supercomputers through MPI or shared memory libraries.
- The video stream uses standard, unreliable IP multicast protocols (e.g. RTP and RTCP). For example, we commonly used 800x600 pixel H.261 video, which requires approximately 300kbps of bandwidth when the image is rapidly changing.
- Audio streams between the collaborators can also use standard, unreliable IP multicast protocols. However, audio requires far less bandwidth than video, but is more susceptible to degredation of quality due to lost packets.
- The communication between the control panels of the collaborators uses both reliable and unreliable multicast protocols. Unreliable protocols can be used for incremental updates of the panels, for example while a user is dragging a slider on the panel. Reliable protocols are used to ensure that all participants are synchronized, for example when the user releases the slider on the panel to set a final value.

This CMT demonstration application uses multiple communication methods all within a single application. However, these are only a few of the communication methods that would be required in a complete collaborative, virtual reality based application. Table 1 is taken

Table 1: Requirements of tele-immersive application.

Flow Type	Latency	Bandwidth	Reliable	Multicast	Security	Stream	Dynamic QoS
Control	< 30 ms	64kb/s	Yes	No	High	No	Low
Text	< 100 ms	64kb/s	Yes	No	Medium	No	Low
Audio	< 30 ms	$N * 128$kb/s	No	Yes	Medium	Yes	Medium
Video	< 100 ms	$N * 5$Mb/s	No	Yes	Low	Yes	Medium
Tracking	< 10 ms	$N * 128$kb/s	No	Yes	Low	Yes	Medium
Database	< 100 ms	> 1GB/s	Yes	Maybe	Medium	Maybe	High
Simulation	< 30 ms	> 1GB/s	Mixed	Maybe	Medium	Maybe	High
Haptics	< 10 ms	> 1Mb/s	Mixed	Maybe	Low	Maybe	High
Rendering	< 30 ms	> 1GB/s	No	Maybe	Low	Maybe	Medium

from [Ref: 9 flows paper]. It outlines nine distinct types of communication flows that are required in such an application, along with the characteristics and requirements of each flow. In addition to the basic nine flows, each flow may have additional requirements placed upon them, such as varying levels of security, compression, and network quality-of-service.

3 Nexus

The previous section outlines the wide variety of communication modalities that must be simultaneously supported in an advanced network application such as the CMT collaborative analysis and visualization system. In general, the method used for a communication can vary according to where communication is being performed, what is being communicated, or when communication is performed [Ref: Managing multiple communication methods in high-performance networked computing systems, Foster et al].

Currently the developer of such an application must program to a variety of APIs for these various flows (e.g. TCP sockets, IP multicast, reliable multicast libraries, MPI, etc), and must know myriad details about each API in order to achieve good performance (e.g. TCP socket buffer sizes). In addition, this burden only increases as these applications add such features as security and network quality-of-service guarantees.

What is required to make the writing of such applications truly feasable is to separate the communication API from the communication method. This is exactly what the Nexus library does. Nexus, the communication component of Globus, has simple, general ways for expressing communication, based on the abstractions of startpoints, endpoints, communication links, and remote service requests. These abstractions are able to express the wide variety of communication modalities described above. The Nexus implementation maps these abstractions on a wide variety of underlying communication methods.

Nexus programs bind communication startpoints and endpoints to form communication links. If multiple startpoints are bound to an endpoint, incoming communications are interleaved, in the same manner as messages sent to the same node in a message passing system. If a startpoint is bound to multiple endpoints, communication results in a multicast operation. A startpoint can be copied between processors, causing new communication links to be created that mirror the links associated with the original startpoint. This support for copying means that startpoints can be used as global names for objects. These names can be communicated and used anywhere in a distributed system.

A communication link supports a single communication operation: an asynchronous *remote service request* (RSR). An RSR is applied to a startpoint by providing a procedure name and a data buffer. For each endpoint linked to the startpoint, the RSR transfers the data buffer to the address space in which the endpoint is located and remotely invokes the specified procedure, passing the endpoint and the data buffer as arguments. A local address can be associated with an endpoint, in which case startpoints associated with the endpoint can be thought of as "global pointers" to that address.

An advantage of the startpoint construct in a distributed computing environment is that the startpoint can be used to encapsulate not only information about *where* communication should be performed, but also *how* to communicate. Different communication methods can be associated with different communication links, with selection being either automatic or user guided. Currently, Nexus supports the following communication methods:

- **Local:** reliable ordered unicast within a single process
- **SysV shared memory:** reliable ordered unicast between processes on the same computer
- **MPI/MPL/INX:** reliable ordered unicast between processes on different nodes of a single distributed memory computer cluster
- **TCP:** reliable ordered unicast
- **UDP:** unreliable, unordered or ordered unicast
- **IP multicast:** unreliable, unordered or ordered multicast
- **XTP:** reliable, source ordered multicast
- **Totem:** reliable, totally ordered multicast

In addition, a message transform, or filter, can be applied to each communication link. This allows operations such as compression, encryption, and profiling to be specified and performed on a per-link basis.

Our experience is that the Nexus abstractions nicely capture the wide variety of communication methods, and cleanly map onto a wide variety of underlying protocols and capabilities (e.g. security and QoS). However, when trying to use the Nexus API for the wide variety of communication methods, we found that one part of the API is still not sufficiently generalized. The APIs for creating startpoints and endpoints is currently better suited for the creation of unicast communication than multicast communication. In particular, there is currently no way to directly bind a startpoint to multicast group. Instead, one must first create an endpoint that is bound to the multicast group, and then bind a startpoint to that endpoint. This can be annoying for processes that only want to send to a particular multicast group. We intend to rectify this by adding the communication link management to the API, and then allowing startpoints and endpoints to directly bind to the communication link. Therefore, multicast communication would be setup by creating a communication link with multicast properties, and binding one or more startpoints and endpoints to that communication link.

4 CIF Communication Library

While Nexus provides a uniform interface to a variety of protocols and messaging libraries, that interface was designed to obtain the highest performance possible. The result was an interface which is extrememly powerful for compilers and advanced programmers, but which lacks simplicity. Unfortunately, not all scientists attempting to build collaborative environments are expert programmers. Thus, an intermediate interface was needed, making the same set of protocols available, but in a package they could easily understand and utilize.

Such an interface was was developed as part of the DOE2000 Collaboratory Interoperability Framework (CIF) project. To make the communication protocols more accessible to collaboratory developers, we created a object oriented interface, known as CIF_Comm, which stresses simplicity. This interface consists of three core classes: abstract connection and listener classes, and a factory class to instantiate them.

The abstract connection class provides a very simple interface for sending and receiving messages. It is from this class that all protocol specific connection classes are derived. As the class name and capabilities imply, each of the protocol specific implementations provide a connection oriented, message passing style view of the communication irregardless of the underlying protocol. By providing this consistent view, applications can switch between different protocols simply by instantiating a different class.

The abstract listener class allows traditional client-server applications to implement server-side functionality using CIF_Comm. Once instantiated, the listener waits for connection requests from remote connection objects. These connection requests are transformed into local connection objects when the application requests that listener supply it with the next incoming connection.

In reality, an application will never instatiate a protocol specific connection or listener class. Instead, it will make a request to the factory class which perform the instantiate on its behalf. To facilitate protocol independence in the factory, all requests are made using URLs. As with traditional URLs used by web browsers, the first component specifies the protocol to be used. This protocol information is used to instantiate the correct connection or listener class which is then passed the remainder of the URL. By using the factory class, an application can easily access a variety of protocols, selecting which to use by simply modifying the URL passed to the factory.

At present, the CIF_Comm interface has been implemented for both C++ and Java. Those implementations currently support TCP, UDP, IP multicast, and Totem. In addition, XTP is supported in the C++ implementation and will soon be supported in Java as well. With these protocols, the application has the full cross product of reliable/unreliable and unicast/multicast communication available to it.

These implementation of CIF_Comm have been successfully used in two applications. The first application was the camera controller software developed by Deb Agarwal at Lawrence Berkeley National Laborartory. The second application was the CIF shared state component which is a fundamental piece of the CMT collaborative software.

While developing the shared state component of CIF, we realized that the standard polling mechanisms to see if a new message of connection request had arrived were insufficient. To rectify this situation, we add an event listening and notification mechanism to the core tools supplied with CIF. With this new mechanism, the application could register an event listener with a particular connection or listener class and expect to be notified when a new message or connection request had arrived. This freed it from having to continuously poll a series of objects, allowing it to continue with other work until it was notified of an event.

5 CIF Shared States

The CMT application had a control panel with multiple GUI components on it such as sliders and buttons. In order to allow any number of users to control these components in collaborative space a mechanism to share state information was necessary. While CIF_Comm provides an easy to use mechanism for sending and receiving bytes it does not provide functionality to synchronize data between multiple processes.

The Shared State component of CIF allows for shared control of abstract states in collaborative space across multiple platforms. The impetus of creating the shared state library was to allow for shared control of "widgets" across different computer architectures. If shared control of sliders, buttons, and other arbitrary components could be established a graphical program running on a high end resource could be controlled remotely from a simpler, more accessible computer. The CMT collaborative visualization application uses the CIF_Shared_State Library to do just that.

Recall that in the CMT application the data is acquired at the APS, but is shipped elsewhere to be reconstructed and visualized. It is the CIF_Shared_State library which enables the scientists to control the visualization remotely from the APS.

While the initial idea was to have shared control of GUI compontents in collaborative virtual space, we found it was best to decouple the value of a control from its visual elements. The CIF_Shared_State library is an object oriented API written in C++ and Java that allows for shared control of abstract states. These abstract states can be implemented as GUI components (sliders, buttons, toggles, etc) or more simply as arrays of data primatives (integers, floating point numbers, bytes, etc.) To create a shared state the user only needs to provide a mechanism for packing and unpacking the current values of it into a CIF_Shared_State_Serial object via convient methods provided by the API.

By creating a means for sharing state information as opposed to control information we were able to create unorthodox visual components that provided no control to the user but were used to display useful information, such as histogram graphs, color bandwidth filter curves, and images of all of the users currently participating in the collaborative session. This layer of abstraction between shared data and visual control also allowed us to couple different visual component packages with the messaging structure.

It was essential that the implementation be platform independent. To maximize platform independencies a Java version of the API was created. However, since there were many high end resources that Java could not take full advantage of, and since many visualization tools, including the CAVE library, are written in C, a C++ version of the API was written as well.

Since the idea behind the program that controlled the rendering remotely was that it could be run on a scientist's workstation, it did not need to access high end resources. To keep everything as platform independent as possible the remote controller was written in Java. Sun Microsystems' swing components were used to interact with the Java version of the CIF_Shared_State API and the user. Specific shared states were created to store state information in shared space provided by the swing controls.

The renderer itself was written specifically for the Irix platform so that it could take full advantage of the graphical hardware provided by the Infinite Reality graphics pipe, and therefore could not be written in Java. However since the CIF_Shared_State library was written in C++ and Java with full wire compatibility, this was not a problem. A set of 3D widgets were developed specifically for the CAVE to be used for controlling the rendering. These controls were coupled with shared states in the same way that the swing componets were coupled to the Java shared states, thus providing shared control of the visualization in collaborative space.

6 The Future: An Integrated Flow Architecture

7 Conclusions

References

[1] Leslie Lamport. *LaTeX: A Document Preparation System.* Addison-Wesley Publishing Company, 1986.

[2] Ree Source Person. A really great journal paper. *A really great journal*, 23(2):728–736, Oct 1998.