An interactive browser for BaBar databases

A. Adesanya

Abstract

Approximately 300 terabytes of data per year will be generated from the output of the *BABAR* physics detector. An Object Database Management System (Objectivity/DB) stores this information via a C++ API. Objectivity/DB tasks performed within the *BABAR* offline environment can be split into two groups:

1. General purpose "browsing" performed by all users to obtain a view of the data stored within federations. Browsing is confined to read-only access.
2. Administrative operations involving reading and writing to/from databases in order to distribute data to other sites worldwide. Use of such operations is restricted.

We require interactive tools to perform both sets of tasks. In addition, we have requirements related to scalability and ease of use. Information needs to be retrieved discreetly, in a manner that reflects the *BABAR* system arrangement. Generic Objectivity/DB tools were not designed to address these demands. This paper describes a distributed, interactive framework that caters for *BABAR* users. An implementation is currently being developed which provides read-only browser functionality to the *BABAR* community.

keywords Java, C++, CORBA, Objectivity/DB

1. Introduction

Particle physics detector experiments produce huge amounts of data. The demands placed on storage systems can be immense. The *BABAR* database group chose to implement its physics event store using Objectivity/DB, a commercially available Object Database System. A software environment had to be produced for the *BABAR* experiment, comprising of C++ language interfaces and support tools. More than 64,000 files will eventually store the physics event objects across several networked file servers. These objects are grouped into related collections and the collections are organized according to the experiment hierarchy. A generic browser is bundled with Objectivity/DB but it does not recognize the *BABAR* structure. This structure provides scalable access to the vast quantities of events, in addition to providing an abstraction easily understood by the users (physicists). Administrators also make use of this logical arrangement, since event collections are used to identify which database files need to be exported to remote sites.

2. Design Overview

The decision was made to develop *BABAR*-specific tools, starting with a browser aimed at common users who wished to view event collections from their networked desktop machines. A client / server architecture would enable data access components to reside close to the file servers, while the front-end could execute on a variety of machines. Java was top choice of implementation language for the client. Its standard library contains a fully featured GUI toolkit with a consistent look and feel. The Java run-time environment was available on all supported platforms and there was the option of executing the client within a web browser.

All database operations take place via Objectivity/DB APIs. Although the vendor supplies a Java binding, it does not (at the time of writing) provide complete access to the physics objects that make use of C++ features such as templates. CORBA (Common Object Request Broker Architecture) is a distributed object communications standard. CORBA Object interfaces are defined using a platform-independent Interface Definition Language (IDL). A compiler processes IDL files to produce platform specific stubs for the server and client objects. ORBs (Object Request Brokers) handle the messaging and the data marshalling that takes place between remote objects executing on different machine platforms. CORBA also defines several standard object services. The most widely used is the NamingService, responsible for storing a directory of addresses used to reference CORBA implementation objects. These references are often referred to as IORs.

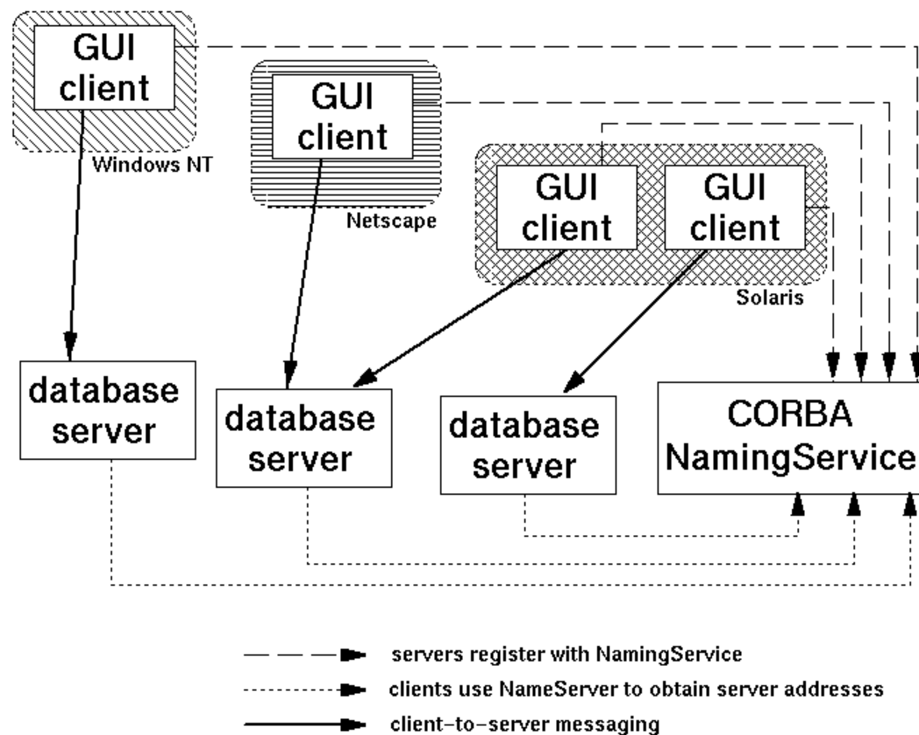

Figure 1: The distributed CORBA components

Figure 1 shows an overview of the distributed CORBA framework. The Objectivity/DB access is handled by the C++ database servers. Only one Objectivity/DB Federation[1] can be accessed from a single process. Users may run individual servers to browse their own private data. Once a server is launched, the address of the implementation object is assigned a logical name and registered to the NamingService. The Java client uses the logical names to lookup a server address used to establish a connection.

[1] The largest entity in an Objectivity/DB system, comprised of up to 64k database files.

3. Scalable browsing

The server does not convert all persistent classes to CORBA interfaces. Instead, it supports a model that maintains the state of the graphical interfaces. The client uses tree widgets to provide an abstraction of the database file and event collection structure. Such trees may well contain thousands of nodes, building the complete trees and passing them across the network to clients didn't seem suitable for interactive browsing. An alternative was to provide access to sublevels of the tree allowing clients to extend their tree widgets on the fly.

Figure 2 shows a screen dump of the Event Collection tree widget. The encapsulated rows are not resolved until the user opens the parent 'folders'. At that point, the client retrieves the required information from the server. The latency is adequate and database transaction time is kept to a minimum. A similar tree is provided for browsing the database files. The link between event collections and database files is established by performing a 'scan'. This identifies the files that store the collection's objects. An administrator may wish to export these files to other Objectivity/DB Federations at remote sites. Event collections contain thousands of events that share the same physics detector conditions. Event collections may also reference other collections. Individual events can be viewed in the browser. The event 'tag' object contains a subset of event object instance data to help improve access speed. A tag filter feature can be used to perform simple queries within a collection, based on tag attributes. Some events are also shared amongst collections and this can be easily identified in the browser.

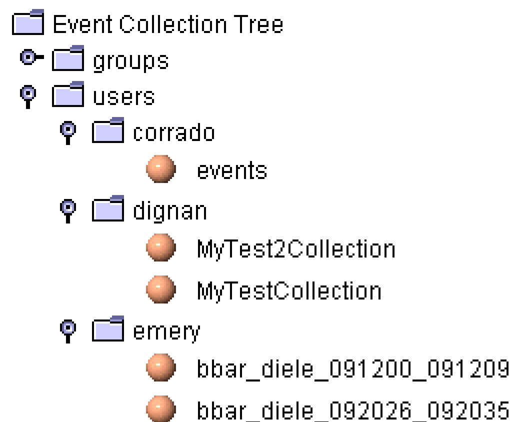

Figure 2: The Event Collection Tree

4. CORBA Interoperability

The current version of the JDK[2] comes complete with CORBA classes and an IDL compiler. For the database servers, a freely available C++ ORB known as TAO[3] is being used. In order for a client to connect a CORBA implementation object (either the NamingService or a database server), the IOR string reference must be converted to a generic object. If successful, the object

[2] Java Development Kit 1.2 is produced by Sun Microsystems.
[3] TAO software and documentation can be found at http://www.cs.wustl.edu/~schmidt/TAO.html

must be cast to the correct IDL type. This process is commonly known as resolving a reference. By default, both the TAO and JDK ORBs attempt to resolve the initial NamingService reference by performing a multicast within a sub-network. However, they do not share the same multicast groups and confusion may arise if multiple NamingService objects are active. Just one NamingService should be visible across the Internet so the current solution is to store the IOR in a text file. This file can be placed in a network file system such as AFS or copied to remote machines. A single TAO NamingService is maintained for all *BABAR* CORBA systems by the computing support team.

5. Multithreaded Performance

Early versions of the browser system performed admirably when restricted to a handful of users. Once the system was released to members of the *BABAR* community, the focus was on scalability. The TAO ORB was written primarily for real-time applications and has an extensive set of options for optimizing performance. The default configuration uses a single thread to process client requests and was an obvious bottleneck. Objectivity/DB supports multithreaded "contexts" and access to multiple cpu hardware provided an additional incentive. Two TAO multithreaded options were tested:

- Thread pool
- Thread-per-connection

The thread pool option makes sense if memory is scarce. An arbitrary number of threads are pre-created and are then assigned to client calls as they arrive. It is worth mentioning that the server is essentially stateless and that client calls are completely independent of each other. Creating a thread for each client connection can consume machine resources very quickly. Each client session has sole use of a thread. As a result, each Objectivity/DB context cache receives a higher number of hits.

6. Conclusions

There is a significant learning curve associated with CORBA programming but the benefits may be worthwhile if the goal is to achieve language interoperability. The client code is 100% Java and this ensures that it should execute within almost any JDK environment including web browsers. CORBA should not be compared directly to low level communications protocols such as sockets. The additional overhead was negligible and only represented a fraction of server latency. Furthermore, the TAO ORB provides methods to disable data marshalling where possible. The distributed framework is a solid platform for *BABAR* browser and administration tools.