Software Advances and their Impact on the Design of Nuclear Data Acquisition Systems

R. Fox
National Superconducting Cyclotron Laboratory
Michigan State University
East Lansing, MI 48824-1321,
M. Nomachi
Research Center for Nuclear Physics
Osaka University
10-1 Mihogaoka
Ibaraki, Osaka,
Japan

September 25, 1996

Abstract

Group communication systems, and several aspects of object oriented software technology were studied in the context of their applicability to nuclear data acquisition system software (DAQ). Some of the technology evaluated can be productively incorporated into the design of data acquisition systems for use at Nuclear Physics research laboratories.

Contents

1 Introduction

Data acquisition systems in nuclear physics have failed to keep up with advances in software technogy. The primary reason for this is the lack of resources in any single laboratory to simultaneously support the running schedule and study how to use new software technologies in future systems. Advances have been successfully adopted in High Energy Physics data acquisition systems.

From June through August 1996, under a Centers of Excellence Grant (COE) grant from the Japanese Ministry of Education, we have been investigating advances in software technology which may be useful for developing the next generation of nuclear data acquisition systems. This paper will describe the technologies evaluated as well as the results of these evaluations.

We present the architecture, of a generic 1990's nuclear data acquisition system in Section 2 . We summarize the criteria used to select technologies and describe the set of technologies actually studied in Section 3 . Section 4 describes several object oriented technologies we studied. Section 5 describes group communications schemes. In Section 6 we show how these technologies impact the reference architecture described in Section 2. In Section 7 we give our conclusions and indicate our plans for future work based on our efforts in this paper.

2 Reference Architecture

Our evaluation of the impact of technology on DAQ systems will be done within the framework of a reference model for a generic DAQ system. In this section, we describe this model. This system, while drawing heavily from UNIDAQ [No94], does not match any single data acquisition system, but contains features which are available in most data acquisition systems.

A data acquisition system is concerned with three major data flows. The flow of event data, the flow of command data and the flow of status data. Event data flow describes how the data which is acquired moves through hardware and software components of the system. Command flow describes how processes within a data acquisition system are controlled, both by people and by other processes. Status flow describes how processes inform people and other processes about their current state, and the global state of the system.

2.1 Event Data Flow

Event data flow takes data from detector subsystems, builds complete events, and distributes this data to processes which record, monitor, and analyze the data. Event data flow can be either internal to a single computer system or, it may cross computer system boundaries. Event data flow is the time critical part of data acquisition systems.

The reference architecture for the data flow part of a data acquisition system is shown in Figure 1 . Buffer managers provide flexibility and isolation. This data flow architecture is representative of the data flow architectures found in Unidaq [No94], the NSCL data acquisition system [Fo85], DART [Ol93], [Be94], and CODA [Wa93]. It should be noted that all of the objects shown in figure 1 are software objects. Some of these software objects, such as data sources, may form logical connections to hardware.

The items of figure 1 which are inside of the "cloud" represent software in a different computer system. The purpose of that segment of the figure is to demonstrate how the event flow can be extended to a distributed data acquisition system.

Buffer managers allow the data flow to be configured to match the specialized needs of an individual experiment. Buffer managers allow unnecessary parts of the data flow to be removed. For example, in an experiment with only a single detector fragment, the event builder can be transparently removed.

Buffer managers also provide test points to which diagnostics programs can be attached. If a specialized monitor for a detector subsystem is created, it can be attached to that detector subsystem's data source buffer manager. Thus each subsystem can be run independently under test and can also be monitored when the entire system is running.

Buffer managers isolate the details of one subsystem from other logically independent subsystems. For example, if a detector subsystem is modified or upgraded with new and improved hardware, the event builder is isolated from that fact by the buffer manager which stands between it and the event builder.

In current data acquisition systems buffer managers operate efficiently by passing pointers to buffers in shared memory regions. The Transmitter process in Figure 1 provides a mechanism to get event data across system boundaries. It feeds a Receiver process which acts as a data source for a buffer manager in the remote system. This allows additional consumers in the remote system to be insulated from the fact that they are located in a system which is remote from the primary data source.

2.2 Command Flow

Command flow is the primary control flow for data acquisition systems. Command flow describes how an individual, program or human tells the system or a part of the system what to do. Our model of command flow is shown in figure 2 Similar methods are used in UNIDAQ [Ba95]. CODA uses a state machine to describe the legal run control transitions of the system [He94]. The ZEUS ZMP message passing system is quite close to this model as well [Mi94].

The main issue in command flow is one of command targeting. Command targeting is the process of determining which process or, in the case of a collective action, set of targets, must be sent an individual command message. In our reference model, this is managed via a Process data base and target selection subsystem.

Commands are issued to the Target selection subsystem. The target selection subsystem queries the process database to determine who receives a command

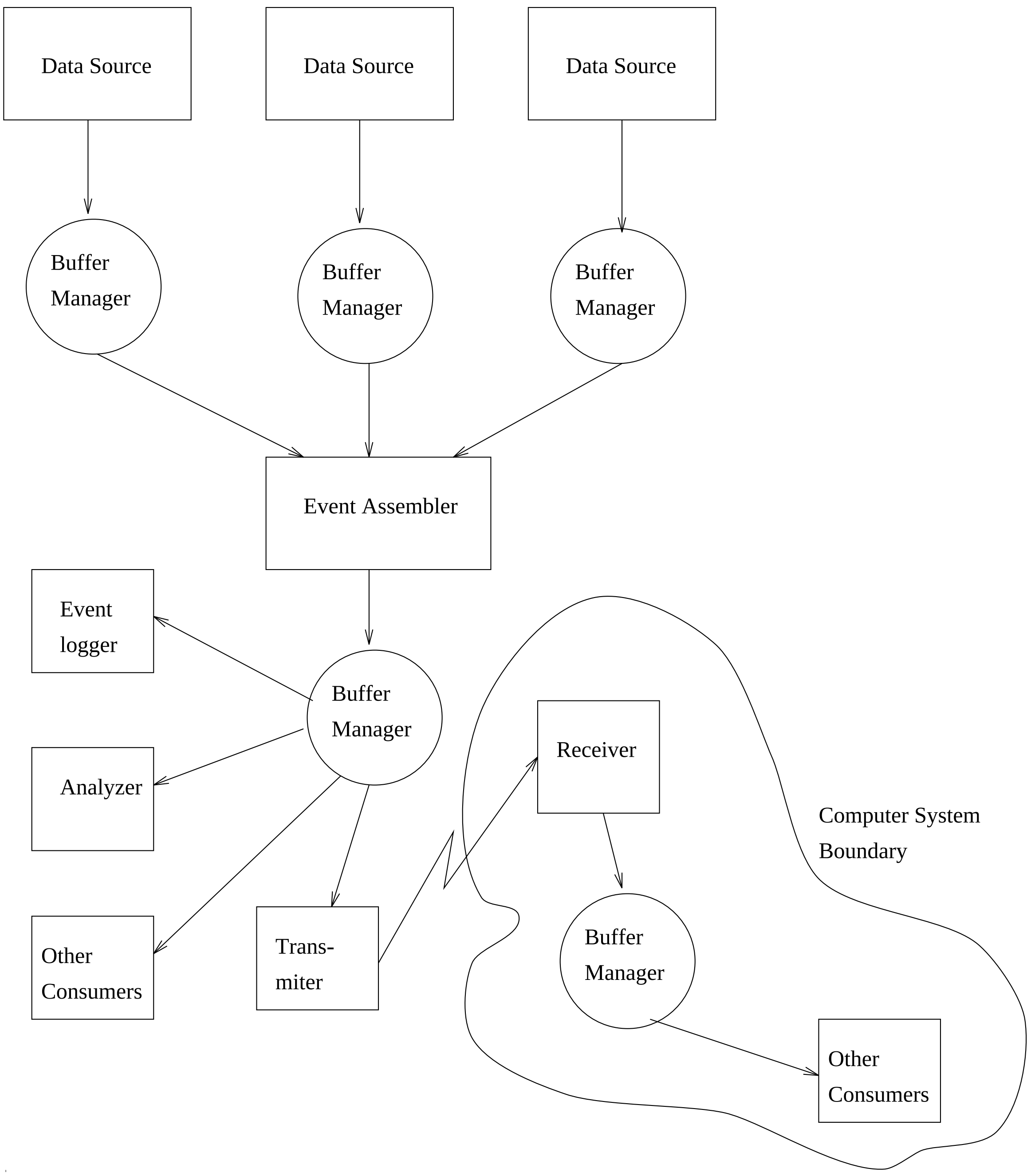

Figure 1: Event flow in a Data Acquisition System

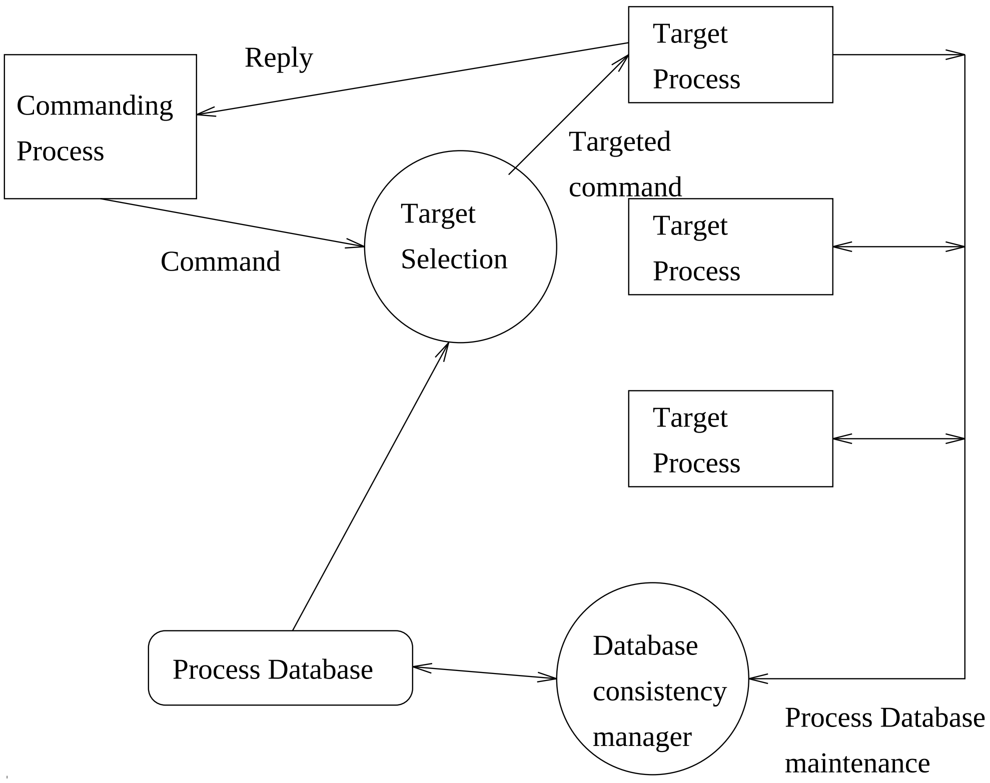

Figure 2: Command flow in a Data Acquisition System

and how it should be transmitted. Commands are then sent to the appropriate process which eventually replies to the original commanding process.

The process database is maintained both by the target processes and via a Database Consistency manager. Target processes register their existence with the Database Consistency manager which enters them in the Process database. The Database Consistency manager is responsible for determining when a process is dead and appropriately modifying the process database. Process liveness is defined as the process existing and responsive to the Database Consistency manager (Watchdog checking).

If, as in CODA [He94], messaging is replaced by remote procedure calls, the port mapper or global/local location manager daemon (depending on whether the RPC system is SUN ONC or DCE derived) serves as part of the process database and database consistency manager. Use of RPC does not eliminate the requirement for routing databases in the case of collective actions, and the consistency of these databases must still be maintained to support graceful system degradation in the presence of process failures.

2.3 Status Flow

Status flow describes how a process lets the rest of the system know what it is doing and what its visible state is. Status flow also includes the logging of exceptional events. For the purposes of this paper we distinguish between two roles in the status flow. A status provider is a process which maintains some status variable. A status monitor is a process which is interested in knowing the values of status variables. We do not distinguish between logging and status propagation, a log message can be modeled as a textual status variable.

The broad range of rates in status modification and monitoring, imply a need for several status distribution mechanisms. Three status distribution mechanisms have been used in the past, Push, Pull and Shared. Some systems have only implemented a subset of these three methods, others have implemented all three.

The graph of figure 3 shows when each of these status distribution mechanisms is appropriate. Push, distribution requires the status provider to actively push status items to some status database or to some set of status monitors. Pull distribution requires the status database system or status monitors to explicitly request the current status item whenever it is required. Shared status places a portion of the status database in shared memory where it can be directly accessed by monitoring processes in the same computer system.

In our reference model we implement all three mechanisms so that we can evaluate the ability of technological changes to simplify status distribution mechanisms as well as the database look up associated with status distributions. Figure 4 shows the reference model for the status flow. Logging is not distinguished from status flow. Logging can be thought of as a pushed status item where the value pushed is a character string containing the most recent message to be logged.

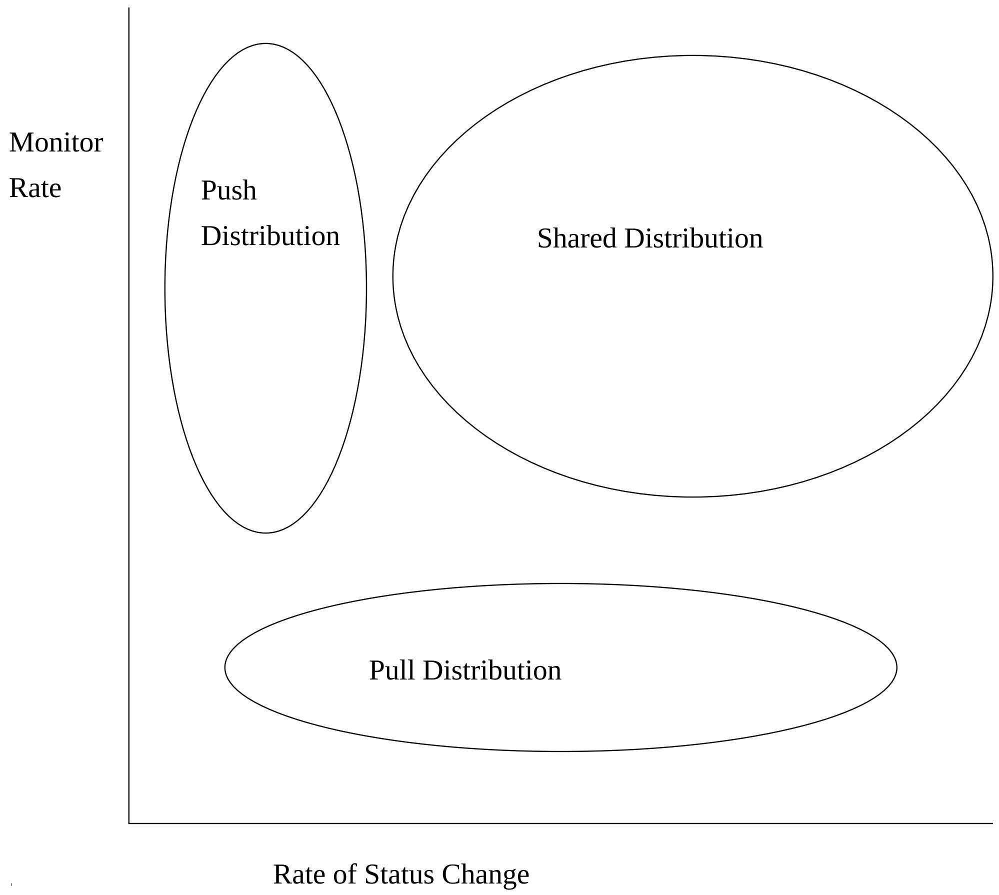

Figure 3: Status Distribution Mechanisms

The right hand side of the graph in figure 3 shows a single process providing a pushed, a pulled and a shared status item. A database of status items provides information describing each status item. A database of status monitors lists the set of status monitors. The monitor processes on the left are entered in the Monitor database. The status items provided by all status providers are entered in the Status Database. Consistency monitors analogous to those used in the databases which support command targeting destroy status items and monitors which no longer exist.

Pushed status items are transmitted to all monitors which are in the status monitor database. A refinement allows processes to register interest in pushed items. Status monitors receive pushes and update their cached values for the status items. When it needs to retrieve a status item, the status monitor determines from a database look up in the status database, that the item is pushed and retrieves the cached value. The cache must be primed on startup through a transaction which is identical to the pull procedure described later in this section and not shown in figure 4 for simplicity.

A pulled status item is maintained internally by its status provider. If a status monitor wants to retrieve the current value of a pulled item, it does a database look up on the item. The look up indicates that the item is pulled and where to pull it from. The monitor issues a pull request from the provider which responds with the current value of the status item.

A shared status item is updated in a manner similar to that of Pulled status items. The new value is placed in shared memory. The monitor process interested in accessing a shared status item looks up the item in the status item database determines that it is shared, and where in shared memory it is, before it 'pulls' that item from shared memory.

Various server systems can be used to convert shared memory items to pulled items for processes which are not local to the system. The push and pull mechanisms can be made to transparently cross systems boundaries.

RPC can be used to distribute status as well. Push or Pull semantics are supported in RPC depending on whether or not status providers or status monitors are the servers.

As for the command flow, the key to a good status flow is to build a good, consistent status database and maintain it in the presence of process exits and failures.

The status flow in this model is based on that of UNIDAQ [Ba95]. DART [Ol95] uses a unique combination of shared memory, delayed pushing and process pulls to manage status variables and statistics. Providers place data in shared memory which is periodically pushed to the *damp* server. The damp server is a centralized access point for both fixed data about the configuration of the run and the changing status values of the run. Access to the damp server is via pulls on the part of the monitor processes.

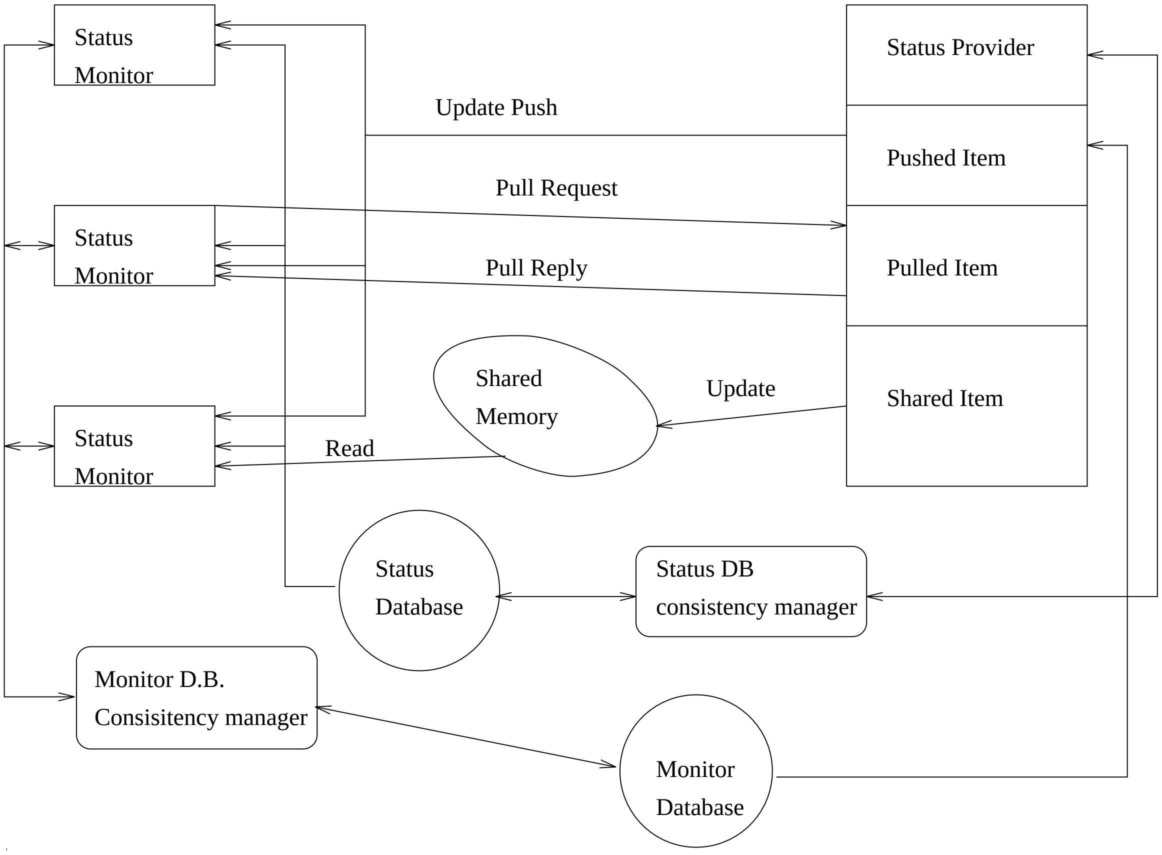

Figure 4: Status flow mechanisms

3 Technologies Studied

This section summarizes our criteria for selecting technologies for evaluation. One point that we would like to establish is our view that the event data flow problem, while placing high demands on the hardware of a DAQ systems, is essentially a solved problem from the point of view of software. We therefore focused on technologies which can help improve the command and status flow. We will pinpoint the problems that we would like to solve and then summarize the technologies which were evaluated to provide partial or total solutions. Sections 4 and 2 will give more detailed descriptions of the technologies evaluated and the results of our evaluations.

While it might seem that the event data flow would be the best candidate for improvement, in fact software for this flow has been well optimized. Copy free buffer managers have been in existence since the mid 1980's [Fo85], [Wh85]. These buffer managers reduce or entirely eliminate data movement associated with routing data from producers to consumers within the same processor.

Figure 5 shows the operation of a typical buffer manager. There are two sorts of clients. Producers and consumers. For the sake of simplicity, we show a single producer and multiple consumers. Buffer managers which support multiple producers exist (e.g. [Fo93]). Support for multiple producers does not appreciably alter figure 5.

Producers request a pointer to a buffer in some shared memory region maintained by the buffer manager. Once they have obtained this pointer, they can read data from the data source devices directly into the shared buffer. When the buffer has been filled, the buffer manager is notified of this fact. From then on, no data movement need take place.

The buffer manager responds to the full buffer notification by sending pointers to the full buffer to the consumers. The order in which consumers receive pointers, the level of concurrency supported, and which consumers receive buffer pointers depends on the individual policies of the buffer manager. The two extremes are totally controlled ordering and total concurrency, these are exhibited by Nova [No], and NSCL's Unix ROUTER [Fo93] respectively. When a consumer is done with the buffer, a release notification is sent to the buffer manager which in turn is responsible for releasing the buffer back to the free list.

Copy free buffer managers ship large amounts of data around within a single computer system in a highly efficient manner. The overhead of a copy free buffer manager is constant with respect to the buffer size, allowing the throughput to be increased increasing the buffer size. While it is possible to improve the feature sets of available buffer managers, it is hard to see how software technology could improve on the performance of this part of the system.

We have therefore focused our work on technologies which address command and status flow. The following questions have guided our efforts:

1. How do we make data acquisition systems easier to write? Object oriented software development has been in use in large segments of the software

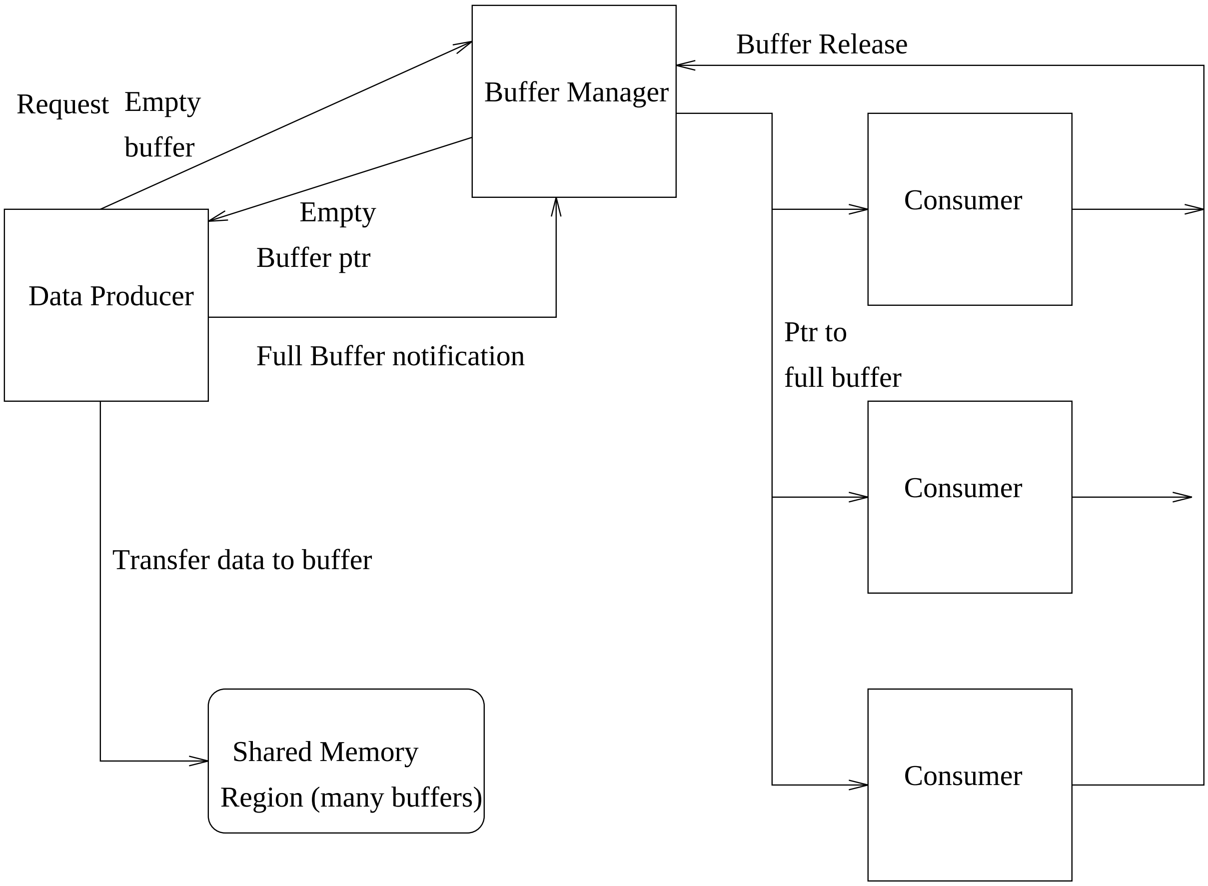

Figure 5: Copy free buffer manager

market. Are there actual productivity gains from this technology and do they transfer to data acquisition system development? Can we provide users with simple mechanisms to integrate their programs into the data acquisition system? Does adopting object oriented software techniques increase or decrease the learning curve for these users and for developers?

2. How do we deal with the maintaining the consistency of databases which describe items exported by active agents of the acquisition system? Do object oriented database systems give us enough power to handle the consistency problem? Do persistent objects represent one way to manage data associated with the experiment?

3. Are there communications systems which offer more expressive power than the point to point systems currently in use? Do distributed objects solve the command targeting issues? How can we handle collective control problems better?

We want to look for unifying themes in our research. We would therefore rather select technologies which solve several of our problems simultaneously than those which only solve single problems.

A problem for laboratories with minimal resources is the time required both to master a technology set, or even to install software supporting that technology set. Since our eventual goal is to implement new data acquisition systems in terms of these new technologies and to make these systems available to several nuclear physics laboratories we wanted to limit ourselves to products that were publicly and freely available.

Section 4 describes our investigation of a few object oriented technologies, including object oriented programming languages, and technologies which are built on top of the object oriented paradigms.

In section 5 we describe collective communications systems. We describe some interesting prototyping efforts which we have done based on a group communication model.

Throughout, our investigations have been oriented towards making working models of segments of data acquisition systems so that we can understand how these technologies could actually be used. When we have selected particular implementations of a technology, convenience has often outweighed technical considerations. We wanted to get a system up and available for test and prototype as quickly as possible. Our choice of a particular product in this study, should not be interpreted as a wholesale endorsement of that product. We must still do detailed product evaluations to choose the best product within the constraints of our development resources.

4 Object Oriented Technologies

Object oriented programming and the object paradigm have been in widespread industrial use for several years. Object oriented techniques are beginning to blossom within the HEP arena as evidenced by [Ar9], and [Br94]. However these technologies have yet to make a widespread appearance in the smaller systems associated with nuclear physics, although one of the authors of this paper has been involved in the production of a visualization tool based on object oriented software development methods [Fo95]. Optimistic claims have been made for object oriented programming as a silver bullet to solve productivity problems and as a route to large scale software re-use.

We begin this section with a discussion of the object oriented programming paradigm. which is described in section 4.1 This review includes a description of object oriented programming and how the object oriented paradigm differs from the procedural programming on which most DAQ systems are now based. We describe a prototype of a segment of the status flow of a DAQ system. We also describe the development of a prototype application framework for data acquisition system programs and show how that can be used to make the construction of data acquisition system software easier for both developers and users.

We continue with a description of persistent object systems. These seem like natural mechanisms to handle some of the database needs of future DAQ systems. We describe some small scale prototyping and indicate where we think current problems lie.

Distributed objects are a natural extension of remote procedure calls into the area of object oriented programming. We describe industrial directions in distributed object systems and indicate some prototyping done with a publicly available near CORBA look alike. We point out some problems with distributed object servers in the DAQ environment

4.1 Object Oriented Programming

This section describes the prototyping which we did involving object oriented development. First the features of object oriented software development are described along with the differences between object oriented software and procedural software. We describe which object oriented languages we decided to evaluate. After describing our prototypes, we give a brief conclusion describing what we feel will be the role of object oriented programming in future DAQ systems.

4.1.1 Features of Object Oriented Software Languages

Object oriented programming languages share several features: Types called classes which encapsulate data and functions; Inheritance which provides a mechanism for producing new classes from parent classes; Run time binding, and polymorphism which provide mechanisms to hide the behavioral differences between class implementations behind common interfaces. Most object oriented languages also support fine grained control of the visibility of class components.

The basic units of procedural programming are data and functions. The object oriented concept of a class encapsulates a data structure similar to a C

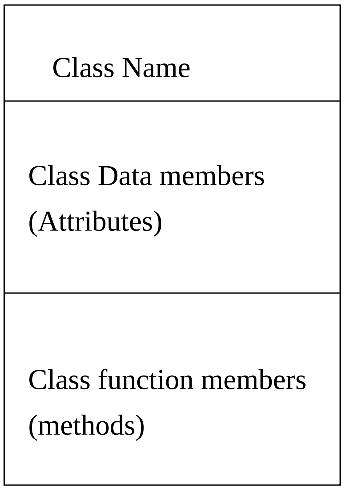

Figure 6: Rumbaugh notation for a class

struct with operations defined on that data structure. These two types of class components are called data members and function members respectively. Data members are also referred to as attributes. Function members are referred to as methods.

The class constitutes the basic unit of object oriented programming. Classes are like types in procedural languages. Objects are instances of classes and are like variables in procedural languages.Given an object which is an instance of some class, the functional members of that class can be called on that object resulting in modifications to the member data items of that object. A snapshot of an object's data members at any given time is sometimes called the object's state. The invocation of a member function which modifies the object's state can be modeled as a state transition taken by that object.

Several object oriented program design methodologies have been developed including those by Booch [Bo91], Shlaer and Mellor [Sh88], [Sh92], Coad and Yourdon [Co91a], [Co91b] and Rumbaugh [Ru91]. We will adopt a modified version of Rumbaugh's notation. We will introduce the features of this notation as needed.

In Rumbaugh notation, a class is described as shown in figure 6 The top part of the rectangle contains the class name. The middle part, describes the data elements of the class. The bottom part of the rectangle declares method functions of the class.

The second major feature of object oriented languages is inheritance. A new class can be defined by derivation from a base class. The derived class inherits all of the data items and method functions of the base class. Figure 7 shows the Rumbaugh notation for inheritance. If a language has visibility control features, it is possible to specify that only a subset of the features of the base class will be available to methods in the subclass.

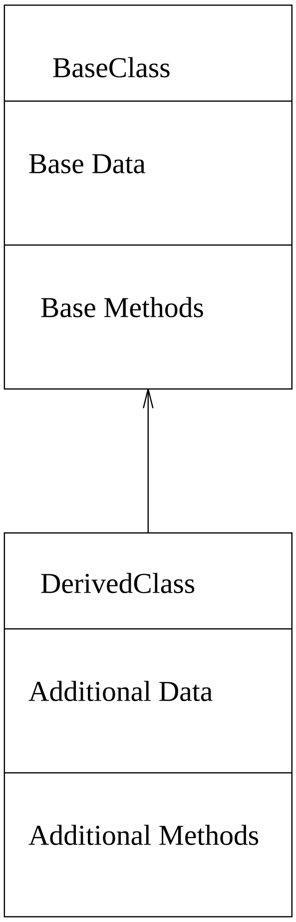

Figure 7: Rumbaugh Notation for Inheritance

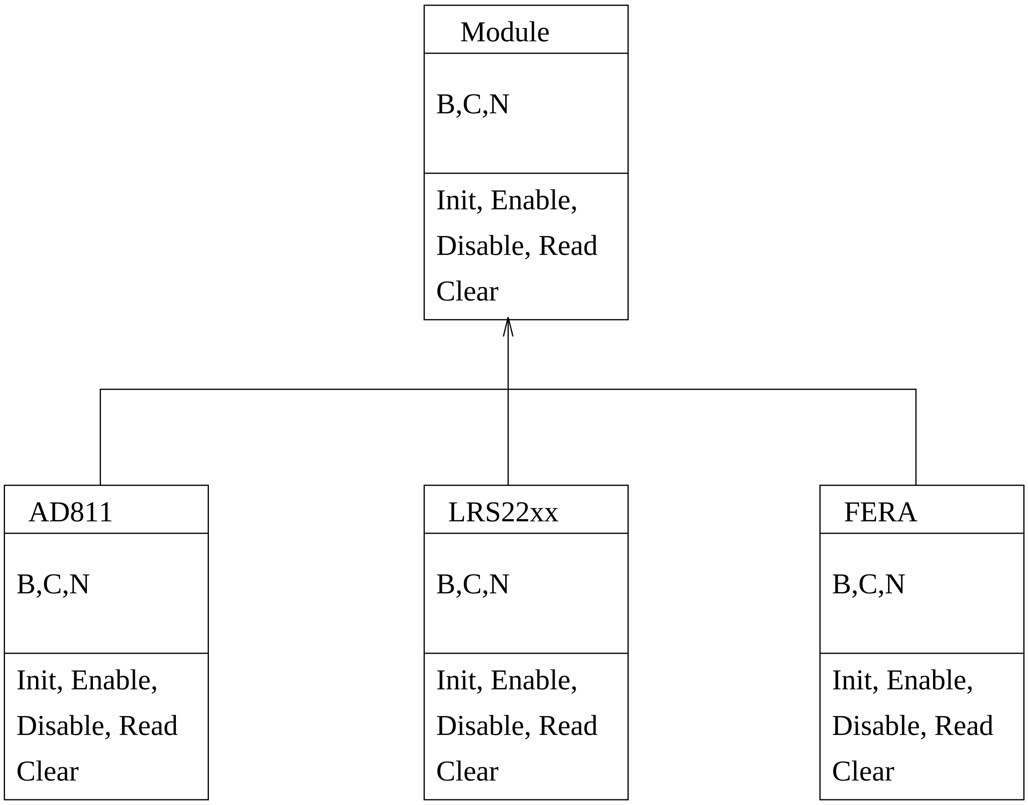

Figure 8: Sample CAMAC Class hierarchy

The arrow in figure 7 points from the derived class to the base class. Since all base data and base methods are inherited by DerivedClass, they need not be listed in the data and method sections of the derived class boxes. Inheritance is used most commonly to express the "Is A" relationship. DerivedClass "Is a" BaseClass with some specialization.

Run time binding and polymorphism are closely related. Polymorphism allows several classes in a class hierarchy to have the same interface to a method with different implementations. The common base of the class hierarchy can be used as a place holder type for an object of one of the derived classes. Run time binding means that at run time, the invocation of one of these methods results in the selection of the implementation defined by the actual class of the object. A closely related concept is that of abstract base classes which are classes that define interfaces for methods, but no implementations.

Run time binding, and abstract base classes are best illustrated by an example. Consider the sample class hierarchy from part of a hypothetical CAMAC readout system shown in figure 8 .

The Module base class provides the data common to all CAMAC modules: The geographical location of the module in CAMAC space (Branch, Crate, Slot). It also provides an interface but no implementation for Initialization, Enable/Disable, Read and Clear functions. The sub classes: AD811, LRS22xx, and FERA provide implementations of these functions which differ according to the data sheets for each module. The Module class is therefore an abstract base class for its subclasses. The subclasses exhibit polymorphism since their instances can all be treated as Module classes. Run time binding of the functions Init, Enable, Disable, Read and Clear allows the Module data type to be used as a generic type for functions which operate on AD811, LRS22xx, and FERA modules. At run time, the appropriate functions are selected based on the object's actual type.

A fundamental paradigm shift is required top do "good" object oriented programming. This shift is what makes the transition from traditional procedural programming to object oriented programming difficult. In procedural programming one builds complete functions which are not modifiable. One programs by stringing together these complete functions into programs.

In object oriented development, a key idea is to build a set of incomplete base classes. One programs by deriving subclasses from these incomplete base classes. Functions are overridden and provided in the subclasses to adapt the inherited behavior of the base classes to the application requirements. Done well, this leads to a much more flexible library base than for procedural programming.

The main effects of adopting an object oriented way of thinking are that:

1. Behavioral information hiding is much easier to express, allowing the production of systems which appear much more homogeneous in the face of implementation differences. The Status handling prototypes will demonstrate this.

2. Re-use units tend to be much larger; patterns of interacting classes and objects, or even entire program structures as will be demonstrated by our application framework prototype. See for example [Ga95] for a catalog of typical interacting class patterns found in object oriented programs.

3. A great deal of up-front effort is required to design the hierarchy of classes properly. A badly chosen set of base classes is just as disastrous to an OO software project as a badly chosen set of data structures and function definiitions are to a procedural project. CASE tools supporting e.g. Rumbaugh like diagraming are very useful. Examination of class hierarchy diagrams can reveal common class features which lead to simplification and generalization of the final class hierarchy. Our experience is that the first class hierarchy design is almost always wrong.

4.1.2 Choices for evaluation (languages)

At present there are basically three well supported choices for object oriented software development languages; Smalltalk [Go89], C++ [St91], and Java [Arn96], [Gos86], [Kr96]. All have public domain or freeware implementations. The syntax of C++ and Java are quite close to that of C and therefore we think that the transition for C programmers may be comparitively easy as only the object oriented concepts need to be learned.

Smalltalk was one of the original OO languages. It is based on a byte code interpreted virtual machine. Smalltalk systems usually feature incremental compilation which in turn supports rapid development and prototyping. However, Smalltalk is beginning to fall out of general use and we feel that its future is limited. In addition while Smalltalk's garbage collection feature, ordinarily be a boon to program development, can lead to severe, unpredictable performance problems in time critical applications.

It turns out that one of the highest quality C++ compilers is the freeware GNU g++ compiler. This compiler has been ported to a wide variety of platforms and is easy to bring up. C++ is at this time probably the most popular object oriented language, however it is not without problems. The C derivation, for example makes unsafe coding possible. Furthermore, implicit generation of object copies along with their implicit calls to copy constructors make it easy for the unwary to write highly inefficient code.

Java is available as licensed freeware from Sun Microsystems. One can also get Java Development Kits (JDK) from CD-Rom's bound into books which describe the Java language. At present platform support for Java is rather limited. New ports will certainly emerge with time. Additional concerns about Java include:

1. Java is implemented as a byte code translator interpreter. This makes it portable but also gives a performance hit which may not be acceptable for high performance segments of the system. Native compilers may well be produced eventually. Just in time compilation systems are starting to become available now.

2. As with Smalltalk, the free store scheme of Java performs automatic garbage collection. Depending on the system, garbage collection passes may be inserted by the run time system in an uncontrolled manner which can result in unexpected and unacceptable delays in execution.

3. Sun has recently moved in a direction which indicates that it may restrict free access to Java implementations, or ports. This may make freeware Java implementations difficult to acquire for laboratories on limited funding.

4. There are implicit 32 bit assumptions in the definition of the virtual machine that Java runs on. These have made ports to 64 bit hardware such as Digital's Alpha, and the MIPS 10K difficult to complete. With physics systems increasingly moving towards 64 bit hardware this must be addressed.

Advantages of Java include a rich library of classes, a high level of compiler support for "safe" programming, classes for threads and communication as well as user interface libraries which provide portable interfaces to native windowing systems and web browsers as GUIs. The applet programming environment includes tight enforcement of security in a computing environment assumed to be unsafe. When run on a system which supports dynamic linking, Java allows the programmer to specify interfaces to software shared libraries written in languages other than Java permitting a bridge to legacy code.

We chose to do our prototyping in C++ using the g++ compiler. This compiler was available on the systems we were using for prototyping, Java was not, an operating system upgrade would have been necessary to support Java. Our choice of C++ should not be interpreted as a rejection of Java. Java is an interesting language and may well serve as the implementation language for parts or all of future object oriented data acquisition systems. Our choice of C++ was based both on convenience and consideration of the problems which must be solved before Java can be used in time critical applications.

4.1.3 Prototypes produced

In this section we describe two prototypes which were produced using the g++ compiler and object oriented design principles. The first prototype illustrates the behavior hiding capabilities of OO, and is a prototypical implementation of a data acquisition system status path. The second prototype illustrates large scale re-use and is an application framework which supports both status providers and status monitor programs.

Status Path prototype The purpose of the status path prototype was to see to what extent the three different mechanisms of status transport described in section 2.3 could be hidden from applications in a data acquisition system. The prototype illustrates the use of polymorphism and inheritance to hide the difference in implementation behavior in the transport of status items from application level code.. For simplicity, this prototype was confined to integer status items. Complete transparency is impossible, since at some point, a choice must be made about how a status item will be transported. We decided to place the responsibility for this choice on the status provider. This led to a class hierarchy for the provider shown in figure 9.

The base class; IntProvider provides a named entity and interfaces to set and get the value of a status item. The subclasses implement the mechanisms of Push Pull and Shared integer status values from the provider's point of view. They provide implementations of the Set/Get functions appropriate to the status transport mechanism. They also require additional information such as the status value in the case of PushProvider and PullProvider, or a pointer to the status item as well as information about the shared memory region in which the status item is located in the case of ShrProvider. The Set method takes care of broadcasting Pushes to monitors, and dereferencing valptr in the case of shared memory items. This class hierarchy exhibits polymorphism which in turn allows the status provider to confine all knowledge of the transport type to a single location (object instantiation or declaration).

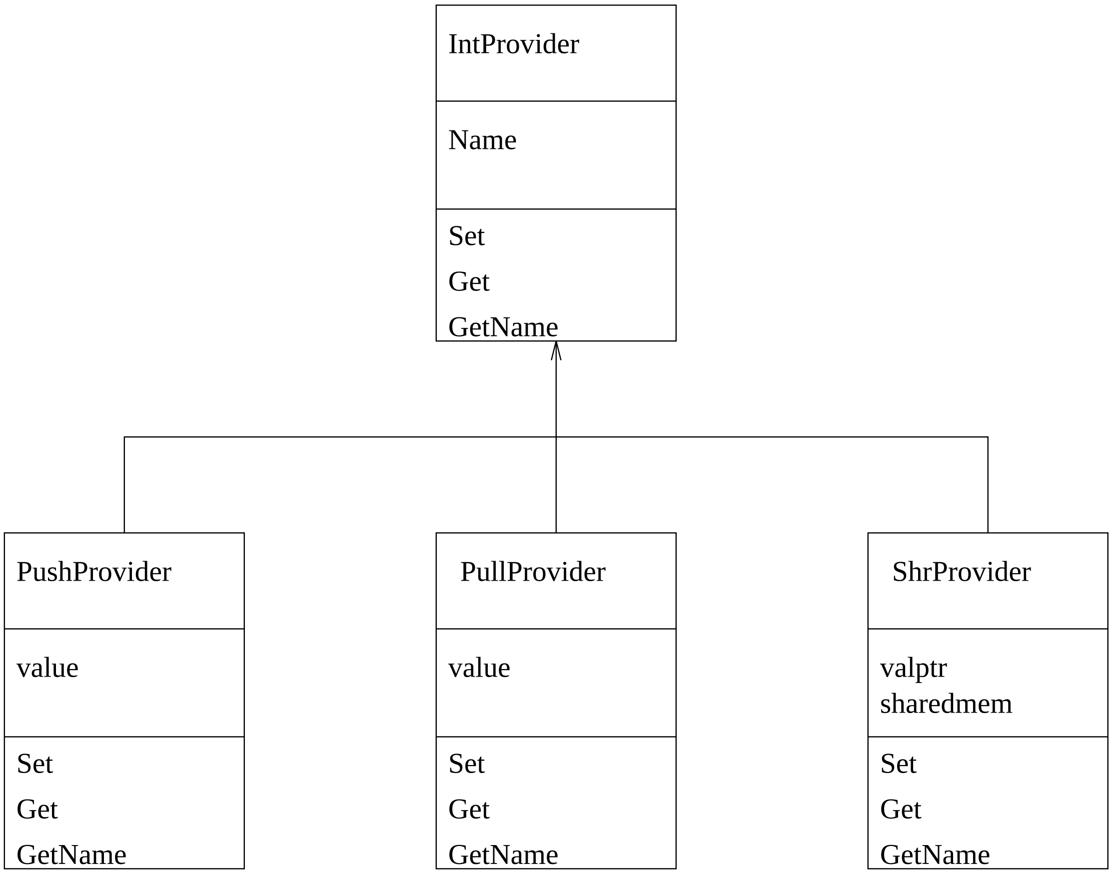

Figure 9: Status Provider Class Hierarchy

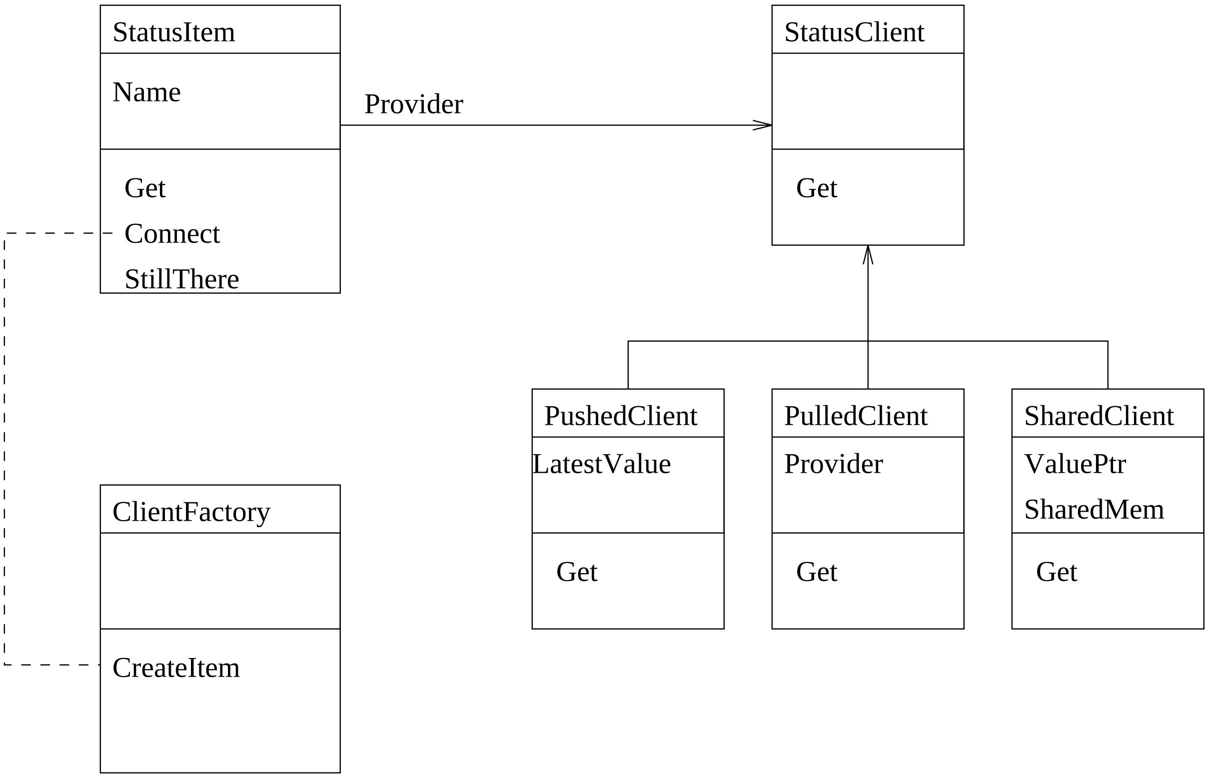

Figure 10: Status Client Class Relationships

In C++ one can use operator overloading to make the IntProvider class appear as if it behaves like an ordinary integer. In that case, the various assignment operations are implemented in terms of the Set interface to ensure that they are bound to the Set function of the actual object's type. Other languages, such as Java do not allow operator overloading. There is great controversy over whether or not the natural notation of an overloaded operator is worth the potential confusion that it can cause.

A simple database was created to maintain a correspondence between status item names and transport mechanisms.

The interesting part of this prototype is the status monitor which is able to completely hide the type of a status item from the monitor software. The object hierarchy used to do this is shown in figure 10

To understand this diagram, we must introduce one more feature of Rumbaugh notation and an extension. The solid labeled line between StatusItem and StatusClient is a relationship. It indicates that objects of type StatusItem contain one instance of an object of type StatusClient. This illustrates a second mechanism for forming classes from existing classes; composition. Composed classes contain data elements which themselves are objects. Composition can be thought of as layering. StatusItem objects are layered on top of StatusClient objects.

The dashed line connecting StatusItem and ClientFactory indicates a collaboration. It indicates that at some point, Connect will interact with a ClientFactory object by calling its CreateItem function. Relationships are a standard feature of the Rumbaugh method, while the collaboration notation shown is similar to an extension we have first seen in [Ga95].

There are two forms of behavior hiding that are accomplished by this class hierarchy. The first is represented by the set of classes derived from StatusClient. This is run time behavior hiding. Polymorphism hides the actual mechanisms of the Get functions.

The second stage of behavior hiding is instantiation time hiding. A monitor class must not explicitly declare a StatusClient as that would create a behavioral coupling between the monitor and the provider. The decision about which type of StatusClient object to create must come from the status item database. If this look up can be done transparently then this is better still. Therefore, we introduce the classes StatusItem, and ClientFactory. All of the procedures required to map a status item name into a particular instance of a status item of type StatuClient are embedded in the CreateItem method of ClientFactory. The construction function of StatusItem calls it's own Connect function which in turn instantiates a ClientFactory and calls its CreateItem method. That call returns a reference to a StatusClient with the appropriate actual type (PushedClient, PulledClient or SharedClient). The Get method of the StatusItem class then simply calls (delegates responsibility to) the Get method of the StatusClient returned by the Connect call.

The ClientFactory transparently maps shared memory status items located in remote systems into Pulled items. All provider classes and programs have been built to handle pulls from all types of status items so that remote processes can access shared memory items and so that a PushedClient can at instantiation time request a pull to initialize it's cache.

Instantiation time behavior hiding is a very useful tool in systems with similar objects which must be selected according to run time criteria. ClientFactory is the only class aware of the fact that there are several ways to transport status. Support for a new status transport method (e.g. reflective memory status) can be transparently added to monitor programs source code by simply adding the new status item class and modifying the ClientFactory implementation (either through inheritance or through direct modification of the Factor class).

The concept of a Factory object is a key design re-use pattern in object oriented software design as described in [Ga95]. In our case, we have also simplified the structure of the system as seen by a hypothetical user by hiding the existence of the factory behind a Facade pattern (StatusItem). Facades are classes which serve as proxies for underlying classes which must be invisible at some level of the program (in our case the StatusItem class).

The StillThere and Connect methods support detection of the death of status providers as well as attempting to reconnect to new status providers which might have come into existence since the status provider died. This also allows for a status provider to be restarted with a status item exported using a different mechanism without the status monitor being aware of this fact.

Application Framework Application frameworks attempt to reuse the design of entire programs within a specific application domain. The use of application frameworks is not new, nor is it limited to implementations in object oriented languages. Examples of successful application frameworks include MacApp [Ap96], the Xt toolkit [Ny92], [Fl92], Microsoft's Visual languages (Visual C++ and Visual Basic) and Unidaq's program template [Ba93].

Object oriented languages, provide a natural mapping via inheritance for the use of application frameworks in specific application programs. Application frameworks usually feature a simple single level inheritance structure. A base class performs system initialization and contains the application main loop. The main loop receives events from the outside world and either responds internally, or calls method functions which can be overridden by the actual application class to provide application specific behavior.

Key to the development of an event driven application framework is an analysis of the events which should be handled by the application framework's main loop. For our prototype we limited ourself to the following events:

stdin input Command input. Representing commands to the program. These are parsed by the framework and passed to command handlers which are registered by the programmer in their application class constructor. Output from the user's command handler is sent to stdout. The application framework therefore provides command line parsing and a mechanism for associating command keywords with command handlers. net cmds.

StatusLookup Represents requests from remote processes to get information about status items which are provided by this process. This is handled without user code intervention.

StatusPush Represents notification sent to this program in its role as a status monitor that a status value has been changed by a provider. This is handled with no user code intervention. The appropriate pushed status object is located in a process internal object list and updated from the message.

StatusPull Represents a request by another process to get the value of a status item. These are handled without calling user code. Idle Called by the application framework after either the wait for an event times out or an event is handled. Can be overridden by the user's derived class.

In addition to events processed by the application framework, it is important to identify the set of services which are provided to the programmer either via member data and functions of the application framework class or via additional classes. The following services were implemented in this application framework:

Status Provider The set of classes described by figure 9 were made available in a class library to users of the framework. These were interfaced to the internal process database so that creating an instance of a status provider would enter it in a name table and a pull response table.

Status Client The set of classes described by Figure 10 were made available in a class library to users of the framework.

Commands A command processor class was written. Instantiating a command processor enters it along with a keyword into the command database. The command's execution function is called when the first word of a command matches the keyword under which it was entered.

Standard Commands The framework provides standard commands: Begin, End, Default methods are called when these commands are received. Main Loop Manipulation

Control Services The framework provides services to set the event wait timeout, and to add/remove file descriptors and handlers to the main loop.

This simple application framework prototype proved useful enough that it was used in several of the prototypes described in this article. Here, however, we only describe a pair of prototypes, a status provider and a status monitor.

The status provider provided the following status items:

Run Number An integer representing the current run number. This was incremented for each Begin. This variable was a pushed status value

Run State An integer indicating if the run was on (non zero) or off. This variable was updated by Begin and End.

Event Count An integer variable representing the number of events which were acquired (incremented when the run was active in the Idle function).

In addition, Begin and End manipulated the main loops event timeout so that if the run was active, the select() was treated as a poll, while if the run was halted, select() waited without timeout. The effect was a program that simulated status from a simple data acquisition system. The entire application (not including application framework code), is 100 lines of heavily commented c++ or about 60 lines of non comment code. These 60 lines would represent the user level programmer's effort in creating a similar application in a real application framework for data acquisition.

The status monitor created four StatusItem variables:

run_number Connected to "Run Number".

run_state Connected to "Run State".

event_count Connected to "Event Count".

rn Also connected to “Run Number”, to test the application framework’s ability to deal with pushes for items for which more than one instance exists in the user program.

The monitor sets the main loop event timeout to two seconds and provides an implementation of the Idle method which simply printed the status variables on stdout. The StatusItem class was modified to detect the death of the provider and provide an appropriate string from it’s Get method. In addition Get was modified to periodically attempt to Connect to the provider in the event that it was called several times while unconnected.

The effect of this was that status monitor and the status provider can be run in the same system. If the provider is halted, the monitor indicates this even if the ’user’ had not performed any error checking. Starting the provider in another system, causes the monitor to eventually realize not only that the status items are available again, but that the event_count item had to be converted to a pull item. None of this is evident in the ’user’ code which is a trivial 48 lines of non comment c++ code (71 lines of heavily commented c++).

4.1.4 Conclusions

The two prototypes described in this section convince us that object oriented programming methods provide a powerful way to describe software in a data acquisition system. The prototypes show how behavior hiding can be used to completely hide complex implementation differences of similar objects from user level code.

The application framework is a useful abstraction and enables users to safely integrate their own programs within an existing data acquisition system.

A well designed application framework is powerful enough to allow simple programs to be written with very little user level code, but is flexible enough to handle special needs. Our application framework turned out to be flexible enough that we used it for most of our other prototyping efforts. This enabled us to produce quite a few prototypes with very little effort. In short, the careful design of a good application framework allows users to safely and easily integrate their own software into an existing DAQ architecture by customizing an existing working program represented as a base class.

An open question is the choice of object oriented languages. Our choice of C++ was dictated by familiarity and convenience, however Java is coming on strong as evidenced by the enormous number of tutorial books and the support it is receiving for use in the WWW. While the development of DAQ systems should rely on well supported accepted languages and techniques, it is also important to use languages that provide good support for DAQ systems response requirements. In addition it is important to use languages which are supported on a large set of platforms.

Java’s interfaces to web browsers and support for user interface development may make it useful in the context of DAQ user interface subsystems as a more robust replacement for Tcl/Tk which is the current language of choice there.

Interfaces to non Java parts of the data acquisition system can be provided via classes with linkages external to Java on systems where that is supported. However, at this point in time we would not write entire DAQ systems in Java. In the future, depending on how Java develops, it may become more appropriate to use Java.

An additional option which has not received much press but is quietly gathering support is Python [Lu96], [Py96]. Python is an object oriented interpreted language with good interfaces to Tcl/Tk. Phython has the advantages of scripting languages such as Tcl/Tk, but has better readability and maintainability. Python has ports to desktop platforms such as Win32 and MACs. This would allow some of the data acquisition system to extend towards these increasingly powerful cost effective platforms. The core support for structured data types such as dictionaries and tuples make it worth closer examination both as a replacement for Tcl/TK in scripting applications and as a prototyping language or an implementation language for components of the system which are not time critical.

Aided by development tools, language technology evolves almost as rapidly as computer hardware and software technology. Therefore as with hardware technology, we must always keep abreast of current developments to see which can be used effectively in our tasks.

4.2 Persistent Objects

This section describes persistent object technology and simple prototyping work that we have done within the context of a freely available persistent object implementation. We conclude with our assessment of the usefulness of persistent objects in data acquisition systems.

4.2.1 What are Persistent Objects

Persistent objects are a natural extension of object oriented programming to long term data storage. There is not much of a dividing line between persistent object systems and object oriented database systems. Object oriented databases can be thought of as a persistent object system with a query language. For the purposes of this discussion, we will ignore the differences between object oriented databases and persistent object stores.

In typical object oriented programming systems, objects are created within a program but have lifetimes which are at most the lifetime of that program. There are cases, however when it would be convenient to retain an object from invocation to invocation of a program, or to pass an object from one program to another program when the two programs are run at different times or concurrently.

Traditional file systems contain only data. Traditional database system contain data and meta-data, called “schema” which describe or impose a structure on the data in the file. Persistent objects and object database systems add methods to the schema which describe and provide operations on the data stored in the persistent object store. Adding locking primitives to the persistent object store allows safe concurrent access by communicating concurrent processes.

Persistent object systems require that the data in the store be composed of objects which are members of classes. The database schema language, often similar in form to C++ describes the data elements of each class as well as interfaces to the method functions for each class. Code implementing the methods must be supplied before data can be loaded into the store.

Once the schema is compiled, objects can be loaded into the persistent store by generating them within a program and storing them. Objects are retrieved from the store by translating some directory entry to an object reference. The object reference serves as a proxy for the persistent object and can be manipulated as if the object were an ordinary program object, with the exception that modifications can be committed to persistent store according to the commitment policy of the underlying system. Several processes my have references to the same persistent object. Locking policies can be used to enforce transaction like synchronization. This technique allows objects in a persistent store to serve as a low bandwidth communication link between interacting processes.

4.2.2 Choices for evaluation.

One need not go much further than the Frequently Asked Questions (FAQ) [Oofaq] list for Object Orientation to get a large list of persistent object and object oriented database products.

Commercial systems include ArtBASE, EasyDB, GemStone, ITASCA, NeoAccess, O2, Objectivity, ObjectStore, Odapter/OpenODB, POET, Statice, UniSQL, and Versant. Research systems which are freely available include: ADVANCE [Bj88], CLOSQL [Mo91], ConceptBase [Ja92], [Ja95], COOL/COCOON [Tr92], Encore [Fe90], Exodus [Ca90], Machiavelli [Oh89], OBST [Al94], [Sc94], [Uh94], Ode [Arl95], ORION [Ki90], and VODAK [Mu92].

Evaluation criteria for this study focus on the fact that we are interested in the technology and its application to DAQ systems at this stage more than we are interested in particular products. We therefore restrict ourselves initially to freeware products which run on our prototyping platform. Other selection criteria include: Convenience We don't want to have to upgrade operating systems, or install several dependent software packages. C++ bindings The system must be usable directly from C++ in particular g++ so that we need not learn another language to use the system. The system should have been used in some sizable projects so that we know that it will be reasonably stable.

Of the research systems ADVANCE and Machiavelli, introduce their own application programming languages. CLOSQL and ConceptBase, are implemented with bindings to languaes other than C++. Exodus and Ode extend C++ either with their own preprocessors or their own compilers. ORION has been commercialized and is now known as ITASCA. VODAK is intended for use in the publishing and multimedia systems rather than for general computation.investigation.

This leaves OBST. Obst is used as the object store for a large scale software

Computer Aided Software Engineering (CASE) tool development project called STONE. It has been available for some years now and boasts a user group well in excess of 1000 users.

OBST features a C++ like schema definition language. Interfaces between OBST and Tcl/Tk exist but we have not examined them. Schema defined in the OBST schema language are compiled and loaded into the the persistent store which is implemented as a UNIX directory. The schema compiler produces two C++ header files. One for implementors of class methods and one for class clients.

OBST persistent stores are organized as a hierarchy of containers. OBST objects can be located via a hierarchical directory structure which is based on a root directory into which other directories can be placed. Directories are simply persistent objects which are of a predefined class. One locates objects in the store by invoking search methods on directory objects. Multiple entries can be made in a directory for a single object, however on object destruction is the programmer's responsibility to locate and destroy any dangling object references both in directories and within other objects in the persistent store..

OBST features NFS safe locking to support concurrent access to the object store. At present, OBST can only be used in a distributed environment by exporting the directory containing the persistent store via e.g. NFS.

4.2.3 Prototypes produced

The clearest application for persistent object systems is state distribution. There exist status items which might better be characterized as system state variables. System states are status items for which cannot be assigned to a single status provider.

We produced a single prototype using OBST. This prototype explored how to create state items as elements in a persistent store, it also probed how to use object oriented techniques to hide the existence of the persistent store from the client (user) program.

A simplified model of the state of the data acquisition system was created. The system was defined as having a run sate which was one of Halted, Active, Paused. The system has a current run number, a title and a taping state (FALSE or TRUE). Methods allow clients to start, stop pause or resume a run. Additional methods allow the client to set the run title or the run number (if the run is halted), modify the tape state, as well as to retrieve elements of the run state.

One of the interesting features of OBST is that the persistent nature of objects intrudes much more clearly into the client of such objects than it does into the implementors of schema methods. This may be an implementation feature of OBST, however we suspect that it is really a characteristic of persistent object systems. Those systems which either extend C++ or are specifically designed for persistent object stores have the option of making access to persistent objects much more transparent.

We would like to use ordinary object technology to hide from the user level

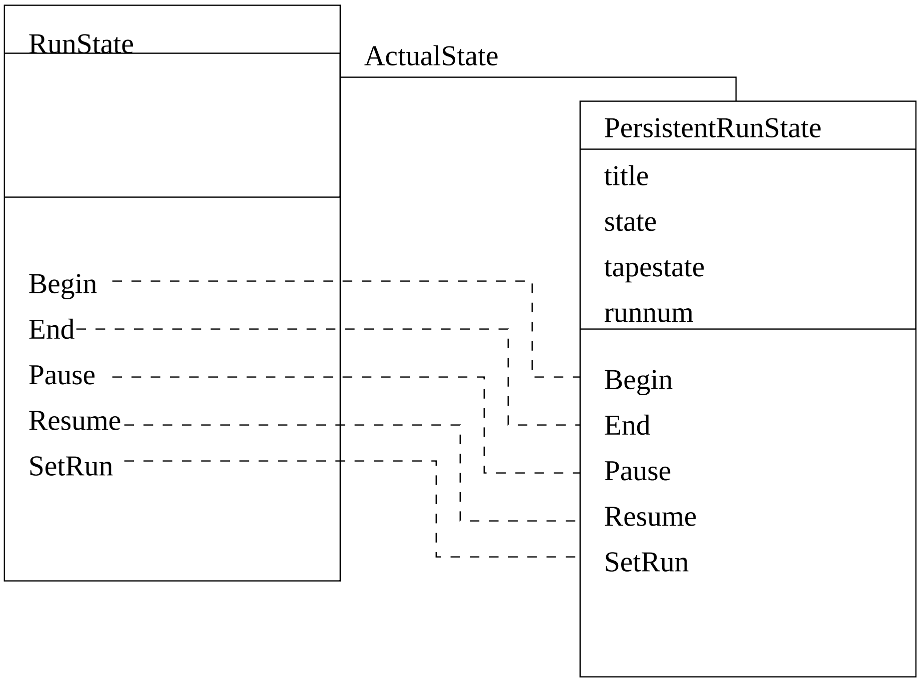

Figure 11: Using a Facade to Hide Object Persistence

client the persistent nature of the object. An object oriented pattern which can be used to do this is the Facade. An Facade stands between the client of an object and the actual objects allowing pre and post processing of requests to be performed before delegating the actual operation to an underlying object. The Rumbaugh notation of the facade class used to hide the persistent nature of our run state is shown in figure 11.

In a facade class, a class is produced which has a set of method functions which provide the final desired interfaces to the actual class. Instantiating the facade object also instantiates an actual object (RunState attaches to the appropriate PRunState object, or creates it if needed). The method function in RunState provide all the OBST specific operations which must prepare for and shutdown access to the actual persistent object which is a member of the PRunState class. In a real system a factory class would most likely be used to produce the facades in order to allow look up errors to be reported to programs wishing to look up existing objects. In our prototype, an attempt to look up a nonexistent object was mapped transparently into an attempt to create a new object of that class.

Based on this facade class, a pair of prototype programs were created. The first a sample run control program could accept simple commands to request changes in a current run state. Errors such as invalid state changes were reported. A monitor program periodically queried the persistent object and printed the current state of the system on stdout. The RunState facade was used in both prototypes to hide the existence of OBST from both programs.

The control program also implemented a simplified mechanism to demonstrate how persistent state can be used to forge a link between data acquisition and data analysis. The run control program supported commands to save the run state into a named persistent object. It could also inspect a previously named run state given its name.

One could imagine the persistent object store as holding data which describes the run, including event file names, tape numbers and methods which might include the event unpacking and analysis code. Retrieval of the run description object could then support replay of the event data. Evolution of the database schema to include subclasses of the run description classes could support stages of analysis. This mechanism of data analysis unfortunately runs right into the versioning problem described in the conclusions for this section.

4.2.4 Conclusions

In the future, distributed persistent object systems may be useful to hold relatively static parts of the data acquisition system's state. These might include run state variables which are handled in our reference model via status path mechanisms. Our prototype work on page 20 describes how this additional form of status distribution could be transparently integrated into the Push/Pull/Shared model.

The problem which persistent object systems may solve is that of state variables which don't have a single clear provider. The state of a run, for example, could be modified by any of a number of processes. Processes may collaborate to manage the run state. The introduction of a run state server is somewhat artificial under these circumstances. In addition, the data acquisition system can be thought of as having state even before processes are started, and after all processes have ended. Persistent object storage systems are one way to implement that concept. For networked data acquisition systems, we would want a truly distributed persistent object store. With OBST, this can only be accomplished either via NFS mounting the repository or wedding OBST to a distributed object manager (see the next section). Some of the persistent object managers we have listed do support distributed access directly, and these deserve much greater scrutiny in future studies.

One important open question in the persistent object community is the versioning problem. The versioning problem arises when the schema for a particular class evolves, or is modified, while objects which belong to that class are already loaded into the persistent object store. References to the old objects are typed with the new schema, but may not contain all the data or method reference information to allow them to be treated without conversion as objects of the evolved class.

Versioning is an important research problem in OODB and persistent object research. Many of the research persistent object systems are in fact proof of concept demonstrations of various ways to handle the versioning problem. Standard methods for treating this from the application point of view must be introduced before persistent object technology can be seriously considered.

In addition standards themselves in OODB systems are still a long way off. However the formation of the Object Database Management group [Odmg] goes a long way towards rectifying this.

One exciting area of application for persistent object stores in the future might be managing the transition from data acquisition to data analysis. As an experiment is run it produces many persistent artifacts. If these, or information about these were held in a persistent store along with manipulation methods, the process of analyzing the data might map well into adding new classes to existing schema and deriving new classes from existing classes to produce analysis stages.

In summary, the concept of persistent object systems provide nice mappings for state variables apart from their implementation as items in the status flow. In the future persistent objects could be used to manage the flow of data from experiment to analysis. At present, there are severe problems with respect to standardization and differing views about the versioning problem. These need to be solved before persistent object systems can be allocated critical functions in a DAQ or analysis system.

4.3 Distributed Objects

Distributed objects provide the object oriented version of Remote Procedure Call. It is appropriate to evaluate this technology to determine if it can be effectively employed in a data acquisition system. This section provides a definition and review of distributed objects. We then describe some of the choices available for systems designers wishing to adopt distributed object technology. Finally we describe our experience prototyping with distributed objects and give our conclusions regarding the use of distributed object technology in data acquisition systems.

4.3.1 What are Distributed Objects

Distributed Object Managers (DOM's) are the extension of the concept of the remote procedure call (RCP) to object oriented systems and programming.

In an RPC environment, one describes a procedural interface using an interface description language. The interface description is compiled producing header files and server and client stub files. The server stubs are filled in to provide an implementation, and a sever program is wrapped around the implementation in a manner which depends on the RPC system.

In a DOM environment, one describes classes of objects using an interface description language. Classes contain instance data items as well as functional interfaces. Classes may also inherit both data and function from base classes already defined. The compilation of the interface description language produces header files and stub files in a manner analogous to RPC. Method development in the server stub files is essentially the same as RPC development. One important difference between RPC and DOM systems is that for each interface definition in a DOM system, several objects may be instantiated. One way to think about the object's data is that it maintains a session context between client and server. This client–server context is absent in RPC. Naturally singleton objects can be produced, and in fact our prototype produces one. The ability to easily hold session level context for distributed objects is one major advantage of the DOM based communication compared with RPC based communication.

All DOM systems require a mechanism to map some object description into a handle to a local object which serves as a proxy for the server's object. In some systems multilayer mapping systems are supported. In the Object Management Group's Common Object Request Broker [Omg95] (CORBA), an object reference can be generated from some string, and converted back into that string. The CORBA specification makes no restriction about the manner in which each implementation treats this string. This can cause implementation problems as the implementation of the DOM is free to choose these strings for the application, making a distributed shared name space the responsibility of the application

The ILU [Jan96] system supports a two level mapping of application specified object names and simple binding strings. An application specified string can be published by an object in a server. It represents a name agreed on by elements of a distributed application which will be used to locate the simple binding string associated with that object. The simple binding string represents an unambiguous specification of the server location and object within the server and can be directly translated into a client object reference in the manner specified by CORBA.

Server objects in turn can create other objects and 'return' them to the client. The returned object is mapped into the creation of a proxy object and what the client software actually receives is a reference to that proxy object..

4.3.2 Choices for evaluation

At this time there are three main players in the DOM standards field. Microsoft's Object Linking and Embedding system (OLE) provides a distributed object manager within the popular Win32 application programming environment. Object Management Group's Common Object Request Broker (CORBA) attempts to define a platform and system independent distributed object system. The OpenDoc [Ap95] system from Apple Computer is a distributed object oriented computing environment based on CORBA. OpenDoc is primarily tuned and positioned for networked distributed compound hyperdocuments. Java has spawned a host of non-standard and standard Distributed Object Managers such as HORB [Hi95]. These are not applicable to our C++ development environment.

At this point, freeware CORBA compliant products do not exist to our knowledge. The closest match is ILU. A Distributed Object Manager from

```
main(int, char**)
{
  TestingStatus stat;
  TestingObjectFactory fac;          // This does the binding for our proxy.

  Testing_T_Logger *log = fac.CreateLogger(&stat);


  if(log) {
    printf("Got a handle\n");
    log->LogEntry(&stat, "Here's a test entry");
    log->LogEntry(&stat, "Here's another log entry");
    exit(0);
  } else {
    printf("Failed to get a handle\n");
    exit(-1);
  }
}
```

Figure 12: Using the logger distributed Server.

the Xerox Palo Alto Research Center (PARC). ILU is subset compliant at the Interface Description Language (IDL) level with CORBA. The rest of the system API includes several CORBA like features. The weakest part of ILU is the application based name publication scheme which is file based. However, see our prototype effort in section 5.3.4

4.3.3 Prototypes produced

We created one prototype. A distributed logger. The distributed logger provides a single sink for log messages which can be sent by arbitrary processes in a distributed data acquisition system. In this prototype, log messages are modeled as simple text messages, and the function of the log server is to place these messages at the end of a file.

The prototype can easily be extended to support a set of log servers, each one represented by a different Logger distributed object. Object oriented techniques were used to hide the mechanisms of object look up and the fact that the logger was a distributed object by using the object factory pattern to generate a facade for the actual distributed object in the client.

Client code to connect to a logger and create a message is shown in the C++ code of the example in figure 12

In this example, the logger object is a singleton, that is there is only once instance network wide of the class Testing_T_Logger. This is hidden in the implementation of the TestingObjectFactory::CreateLogger function. The factory could just as easily take as an argument the name of a specific logging object to create a handle for.

4.3.4 Conclusions

Distributed object technology provides a convenient abstraction for hiding the distributed nature of a data acquisition system from the programs making up that system. Use of distributed object systems is beginning to penetrate the particle physics DAQ field. [Bar96] describes the use of a CORBA compliant Object Request Broker to supply services of an event analysis processing farm to processing front ends and a Fibre Channel based event builder.

The extension of RPC to an object oriented model provides capabilities beyond that of pure RPC where it is difficult to maintain a session level context between server and client. The data associated with an instance of a class (Distributed object) allow state-full distributed systems to be easily constructed.

There are several problems with embedding distributed object systems into DAQ systems. These include point to point communications model, lack of standardization, and inter operability, and loss of application control at the server level.

With the exception of Iona's Orbix+ISIS work [Io95], DOM systems only support point to point communications. Orbix+ISIS is intended to support redundant objects via replication (hot spares). In section V. we hope to convince the reader that point to point communication is a severe limitation, and that collective communications systems provider the power to solve manmy current problems in DAQ system design.

Standardization and interoperability problems exist even within a single distributed object manager specification. CORBA, for example in version 1 does not specify a transport mechanism. Thus CORBA implementations from different vendors may not inter operate. Version 2 specifies an inter operation mechanism, but as yet not many version 2 compliant products are available. Version 2 still does not require an application based object naming scheme although most vendors recognize the need for this.

OLE and OpenDoc, while pushed as general purpose DOMs, are usually cast in the role of mechanisms to distribute parts of a compound document, or produced a window containing information from several applications in a GUI system. This coloring affects the application programming interface in peculiar ways forcing some rather interesting twists to get truly general purpose distributed objects developed and produced.

A major problem of DOM's in data acquisition systems stem from the structure of the server program itself. At some point, the programmer must yield control to a server 'main loop'. This main loop processes network requests for object instantiation, destruction and method invocation. It is difficult or impossible, depending on the implementation to generalize the server's main loop to include other processing which may be time critical (such as responding to event data).

5 Group Communication

Many outstanding problems with DAQ systems stem from the use of a point to point communications paradigm. We evaluated alternatives to point to point communication. The most interesting alternative is a collective communication model called group communication. This section describes group communication, choices for evaluation and a set of prototypes which illustrate solutions faced by data acquisition system architects ranging from message targeting issues to consistency management in distributed databases. Group communication has been studied both in high performance scalable parallel computing, as well as in fault tolerant computing. In message based scalable parallel systems, group communications are used to broadcast parameters to replicated workers prior to handing them individual tasks. In the fault tolerant computing field, group communications are used to maintain groups of replicated distributed processes as hot spares for a primary object.

5.1 What is Group communication

With the exception of the FNAL DART Run control [Ol95], the data acquisition systems that the authors are familiar with are all based on point to point communication mechanisms. The point to point communication paradigm specifies that each message is sent by a single producer process and received by a single consumer process. The group communication paradigm is illustrated in figure 13

Key features of group communication are that:

1. Processes are organized into groups. A group is a collection of related processes which provide a collective service.
2. All processes in a target group receive messages sent by the sending process. (single producer, multiple consumer model).
3. In some implementations, group membership can be dynamically altered.

Buffer managers can be thought of a as a specialized form of group communications, tuned for the efficient delivery of large messages (buffers).

Group communication is an obvious choice as a solution to the message targeting problem, especially for operations which involve multiple processes. In point to point communications, the grouping system still provides a useful process naming scheme. Message targeting is the primary use of group communication in the DART Run control system. We will demonstrate that group communication has applications beyond message targeting.

5.2 Choices for evaluation

Several implementations of software which support group communication are available. The only commercial system we are aware of is the ISIS [Bi94] toolkit which was developed at Cornell University and commercialized by Stratus Inc .

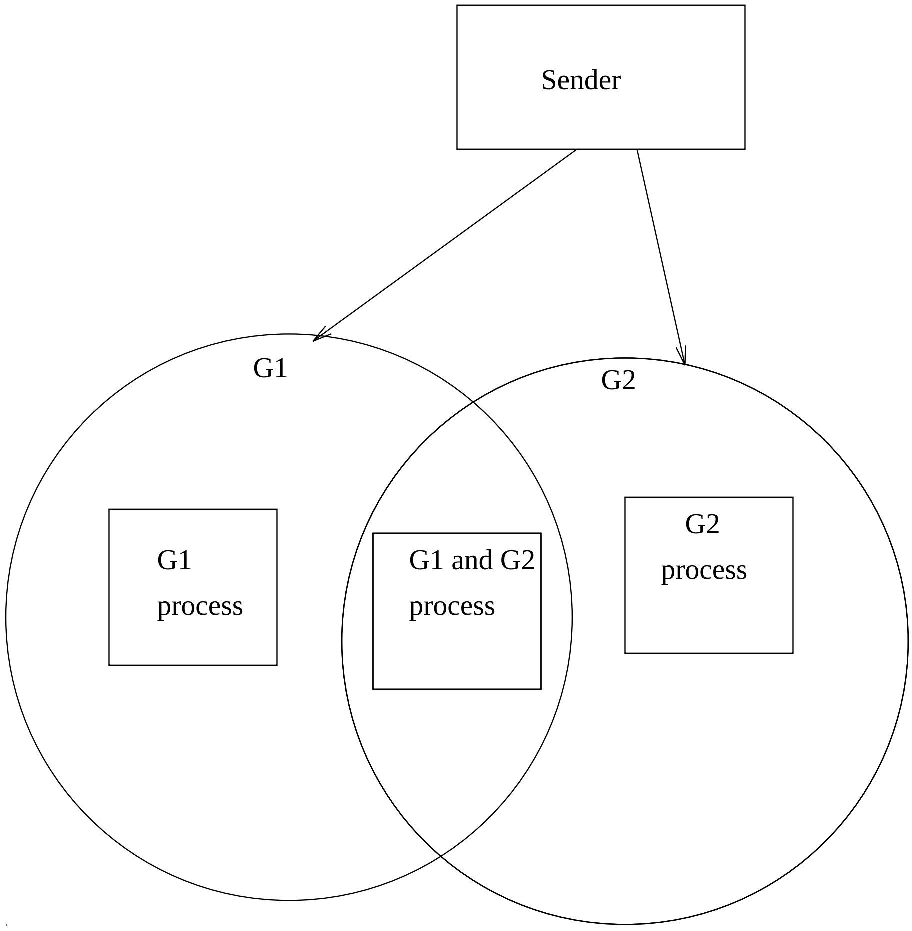

Figure 13: Group Communication

The Parallel Virtual Machine library (PVM) from Oak Ridge National Laboratories (ORNL) [Ge94] is supplied by most vendors of scalable parallel systems and is quite popular in that area. The Message Passing Interface (MPI) [MP94] definition is an attempt to define a portable standard for message based scalable parallel systems. MPI also supports a concept of group communication. In the research area, Transis [Do96], from Hebrew University, and HORUS [Re94] a follow on to ISIS from Cornell are also available.

Our choice of a product to use to explore group communication was based on availability and portability as well as feature sets. Our choice reflects a desire to work with some group communications system which has properties close to what we need and does not necessarily imply an endorsement of that system on our part.

The ISIS toolkit. based on research work at Cornell, is now a commercial system, however it is quite widely used. It should be considered in future work, if the price is right. However the limited duration of this project made it impractical to use ISIS for our prototypes.

HORUS is a light weight group communications system from the developers of ISIS. HORUS is an interesting project. The aim of HORUS is to provide group communications on top of a set of user definable protocols which can be pushed into the connectivity layer in a manner analogous to the way in which STREAMS are built in UNIX SVID. For real systems, this has promise as a mechanism to integrate some of the custom systems which sometimes make their way into data acquisition systems directly into the group communication scheme regardless of the actual mechanism of communicating with them. For the purposes of prototyping, this system was a bit too complex to be easily learned in the time range of our projects.

Transis uses network level multi casting as the transport mechanism for group communication. Transis is freely available from Hebrew University in Jerusalem. Transis makes use of a modified version of the Trans protocol. Trans uses Multicast extensions to UDP to make efficient use of network bandwidth. Transis supports several concepts of message ordering in group communications. However at this time Transis does not support heterogeneous systems in the sense that it does not adopt a concept of network data representation.

MPI and PVM are the two entrants from the scalable parallel processing arena. MPI is a proposed standard for message based parallel computing. However MPI is missing several features that are important for DAQ systems.

MPI has no concept of process control, in the sense that the set of processes cooperating within MPI are intended to be fixed in time (no exits, no new processes), or manipulated via mechanisms external to MPI. Process management is usually provided by shell based process managers supplied with specific MPI implementations in a non-standard way.

MPI process group membership is managed by the cooperating processes themselves. In order to build a group communicator, a processes must already know which processes are in the target group. Our prototypes will demonstrate algorithms for which this is not acceptable.

Finally, MPI's concept of a message is limited to arrays of a single data type.

While heterogeneous systems are supported through a standard for external data representation (XDR), MPI restricts this to a single data type per message. This is a good match to the sorts of communication which usually takes place in distributed parallel environments, where large arrays of data are broadcast to worker processes for computation. This is not a fatal flaw for DAQ systems since most of the information that we propose to send via group communication can be sent as textual messages. Freeware implementations of MPI are available.

PVM overcomes the restrictions of MPI, and is also freeware. PVM has the support of a very large user community. PVM is usually provided in sales of scalable parallel systems on the market today. PVM's group management is server based and is dynamic. Processes actively join or leave groups and the group membership is known to the PVM servers rather than individual processes. A PVM process may be a member of an arbitrary number of groups.

PVM includes some simple process management primitives which allow computers to join or leave the virtual machine. Processes can be started by other processes. Processes can also leave the PVM system via a function call. In addition processes can arrange to be informed of processes leaving the PVM. Process death is noticed by the PVM servers and automatically results in group membership cleanup. Processes may also request a message on informing them of the exit of other processes.

We produced all of our prototypes using PVM version 3. We implemented a two node PVM system based on FORCE Sparc CPU boards running SunOS. The next section describes the set of prototype systems we produced.

5.3 Prototypes produced

We produced several prototypes based on group communication which illustrate how it can be used to solve some of the organizational problems in DAQ systems that were described in the introduction.

Prototypes included message targeting for command flow. Distributed consistent databases, and a prototype status distribution mechanism.

5.3.1 Message targeting for command flow.

Group messaging supplies a name space which directly supports command flow message targeting. Each process in the DAQ system joins a group of one. The group of one provides a network distributed process name space. Messages can be sent point to point via the group communication system by sending them to the appropriate group of one. Collective commands can be handled as well by forming groups of processes which are targets of collective commands.

The Application Framework described in Section 4.1.3 was modified to accept command strings via group PVM messages. Using the application framework concept meant that we were able to make this change without any user code being aware of the existence of group communications with one exception: The application framework's class constructor was parametrized with a group name which represented the unique name of the application within the system.

Application framework code would ensure that this name was unique by joining the group and then ensuring that the group membership count was 1.

The command handling system already passed a reply stream to the command processors. Command processors were expected to write any output on that stream. In the case of a PVM message, a C++ string output stream was passed to the command processors. The command processors functioned exactly as before. The PVM part of the command dispatcher unpacked the string from the string output stream and sent it back to the command invoker as PVM message. Message type ID's were used to distinguish between commands, their replies and other PVM messages used by the system.

We wrote a program within the application framework which takes a command line and a group name and sends the command to that group. Responses received are written on stdout. Once the application framework was able to understand PVM messages and dispatch to user level code appropriately, the command dispatcher required only 62 lines of executable C++ code. The command dispatcher was tested with a framework program which understood a broad range of commands.

In some cases, it is desirable to import software which expects to be run from a terminal interface. Such programs could have a simple shell wrapped around them which stands in as a proxy for the imported program in group communication. The proxy would accept terminal or group commands, feed them to the imported program through a pipe, and send all output from the pipe back to the most recent commanding process (or terminal). Similar proxies can be used to extend the group communications system into systems on which it is not supported.

5.3.2 Distributed database look up in the status flow.

In section 4.1.3 a we described an object oriented implementation of a prototypical status flow in a data acquisition system. Section 4.1.3 described an application framework was extended to produced a status provider and status monitor program. This section describes how group communication was used within that prototype to implement status database look ups.

Figure 14 illustrates the protocol which implements the group look up. Order is given by the numbers on the message arrows. Processes which provide join a provider group. Joining the status provider group is made transparent by hiding that process in the class constructors of the status classes. Each status provider keeps an internal list of the set of status items it exports. In our prototype, these lists were maintained by the status item constructor functions.

When a status monitor wishes to look up the attributes of a status item, it sends a query specifying the object it wants. The query is broadcast to the status provider group. All of the status providers consult their exported status list to determine if they can resolve the query. The status provider which actually provides the status item in question replies to the query. Timeouts detect queries of non-existent status items.

This query system is automatically self consistent. If a status provider dies,

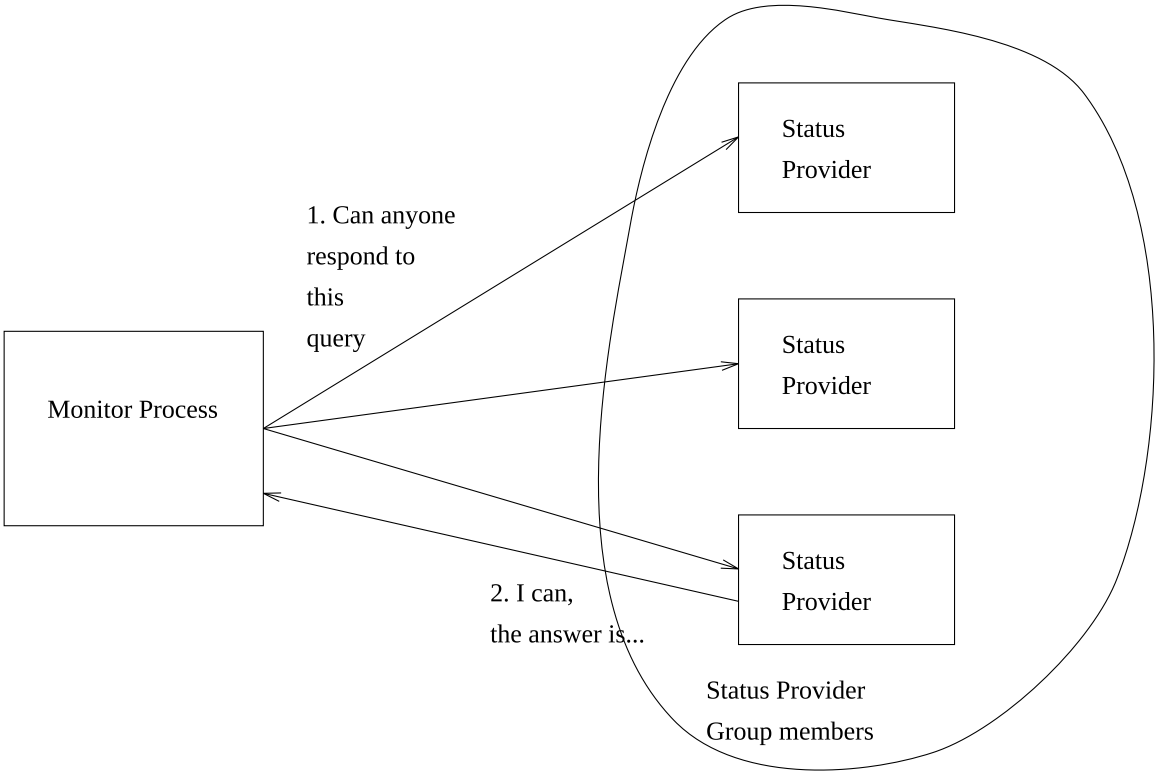

Figure 14: Distributed Database Queries with Group Communication

no cleanup is required beyond what the group communication system does automatically since queries for items which had been managed by the dead process will time out. This in most cases is also true for "live death" since a process in that state is very likely not responding to group communications. This system is scalable, under the assumption that the group communication system provided by PVM is scalable, as the look up is done concurrently across all of the status providers in all of computers in the system. If desired, processes could register with a liveness monitor which in turn could ask the system to inform it of process death.

Naturally the handling of status database query messages is managed by the application framework without the knowledge of user code. The status factory class handles the client side of the database look up transparent to user code.

5.3.3 Distributing pushed data items

Group communication also provides a natural mechanism for implementing extensible and user status monitors. One problem faced in implementing systems which support an arbitrary set of status monitors is deterimining the targets for status pushes. With group communication, pushed status items are handled as follows: All status monitors join the Monitors group. Each time a pushed status item changes value, an update message is sent to the Monitors group. When a monitor connects to a status item and finds that it is pushed, it initially loads the item's value using the pull mechanism.

This mechanism was used to implement pushed status items in the status prototypes of Sections 4.1.3 and 4.1.3. The Push was done transparently as part of the status object's Set method.

A similar technique can be used to distribute log messages to log servers. Log servers join a logging group. Log messages are simply sent to this group.

5.3.4 Improving on ILU's object publication mechanisms

In section 4.3.3, we described the distributed object system, ILU. ILU was used for prototyping a simple logging system. One problem with ILU is that it has a weak application name publication scheme. To publish the mapping of application specific object names to simple binding strings, ILU creates a set of files in a bindings directory. Each file in that directory contains a simple binding string representation of a published object. The name of the file is the application defined name of the object which must be published by user server code.

This binding system requires that the bindings directory be NFS mounted on all systems participating in distributed system. Furthermore if servers exit, ILU as written does not remove the set of bindings files for the objects exported by that server.

Group communication database look ups are a better way to perform name look up. Section 5.3.2 describes a general protocol for database look ups. Modification of the server loop for ILU and definition of new object registration functions allowed ILU to use distributed database look up to translate mnemonic bindings to simple string bindings. This mechanism automatically detects at binding translation time the lack of a server for a previously existing binding.

5.4 Conclusions

In addition to providing a natural solution to the command targeting problem, group communications offer a valuable new technique for dealing with database and process management. In past data acquisition systems, various ad-hoc schemes were used to maintain consistency in item/process/attribute databases. This was necessary because the items described by these databases were in fact managed by processes external to the database system. Group communications bridges this gap by distributing the resolution of database queries to the processes which actually manage these items.

Group communication also supports a collective form of message targeting. That this is useful has been demonstrated convincingly in DART where commands, consisting of TCL scripts, can be targeted at not only process groups, but lists of groups with command rollback implemented via a two phase commit procedure.

The actual choice of a group management system is still unclear. While MPI has many drawbacks for use in data acquisition systems, the likelihood that it will be promoted to a standard in the parallel computing environment makes i a serious contender. PVM on the other hand is widely used. PVM also provides much more appropriate functionality than MPI.

The ISIS toolkit is essentially the only commercial product which is used in this area and therefore deserves a careful look. The development of ISIS+CORBA, a CORBA compliant Distributed Object Manager implemented on top of the ISIS distributed toolkit is also interesting. ISIS+CORBA may remove one of the objections we raised in our evaluation of Distributed Object Managers.

The most interesting feature of the TRANSIS system is that it provides a wide choice of message timing semantics which, among other things, supports transparent two phase message commitment (ensuring all members of a group receive a message before delivering it to process level code).

DART chose to implement their own group communication scheme based on some features of the ISIS toolkit. The implementation of the DART group communication system was done by FNAL however.

Finally, it is clear that the decomposition of the data acquisition system into a set of groups is a nontrivial process. In the case of database look ups, the set of groups and members fall out automatically, and can be maintained by software at levels lower than the user level. For command targeting, it is not clear that the job is that easy. Careful decomposition of system functionality and examinations of the role of each process may be necessary. Hopefully this can be automated outside of user level code.

6 Impact on the Reference Architecture

We have described short term prototyping projects based on object oriented software and collective communication technologies. In this section, we re examine our data acquisition system reference model to see what simplifications can be made the light of our experiences.

Object oriented technologies are mostly useful in the code level implementation of the system. While object oriented software techniques can lead to a clearer picture of the implementation of an architecture, it is not clear that the Object oriented technologies have the power to change the actual architecture of the system. Object oriented modeling, does provide a graphical notation which makes it possible to grasp common elements of a system and to iteratively simplify the design of the system into common base classes. Object oriented software development can lead to simple and clean interfaces to complex parts of the data acquisition system.

Our prototyping work in group communications points the way to simplifications in the command and status flows of the reference model. Figures 2 and 4 the original versions of the command flow and status flow respectively.

These flows are complicated primarily due to the separation of database and service provider. In the case of the command flow, the database of process names is separated from the existence of the process itself. In the status flow, the database of status items is separate from the provider of these status items.

Adoption of group communication for command targeting simplifies command flow by removing a large section of figure 2 The resulting picture is shown in figure 15

The targeting database and associated consistency manager are replaced by a packaged group communication system. Processes join groups appropriate to the set of commands they will receive. In DART, process groups described functional subsystems of the DAQ system. . Groups could just as easily consist of subsets of the command set recognized by the data acquisition system. It may still be necessary to maintain a database of command target pairs, however since targets are now groups, this database can be static, and would not require consistency management.

The figure also illustrates the point that groups of potentially many processes are targeted for command flow. The commanding process will need to be able to query group cardinality to properly process reply messages associated with a command.

We have also described how group communication can be used to simplify status flow in a DAQ system. Figure 16 shows the amended status flow diagram. This figure should be compared with figure 4

Group communications have eliminated all external databases, replacing them with distributed process resident tables. Database queries are taken care of by group distributed queries. Status monitors join the status monitor group when they declare interest in a pushed status item, this entitles them to receive status update pushes. Pushes for items the process is not interested in are ignored, while the application framework transparently updates the cache of the

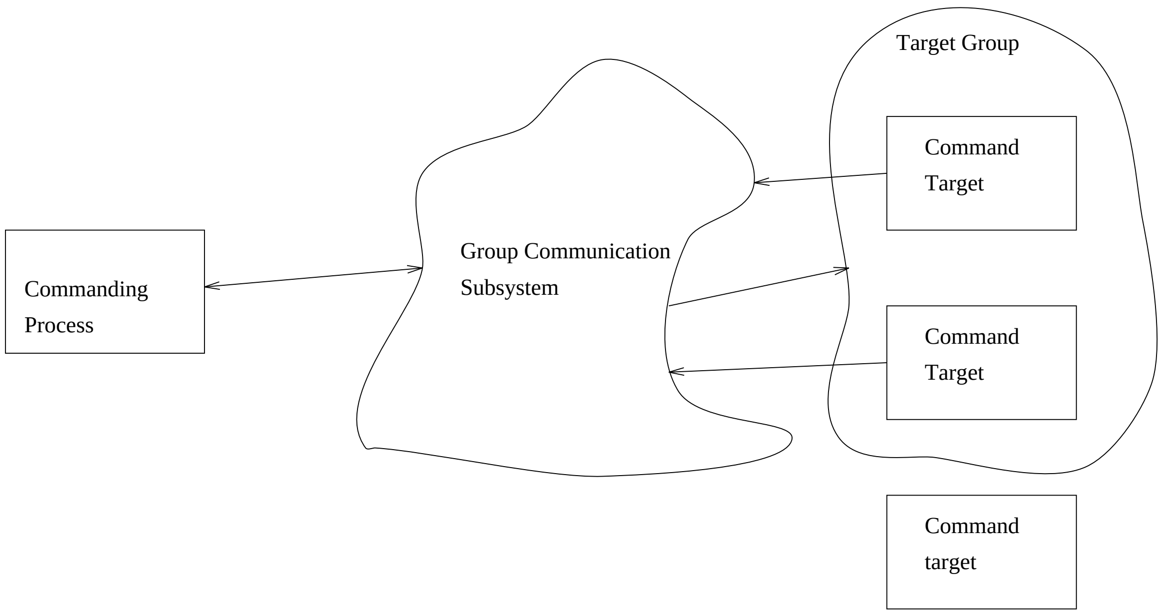

Figure 15: Command Path Simplified through group communication

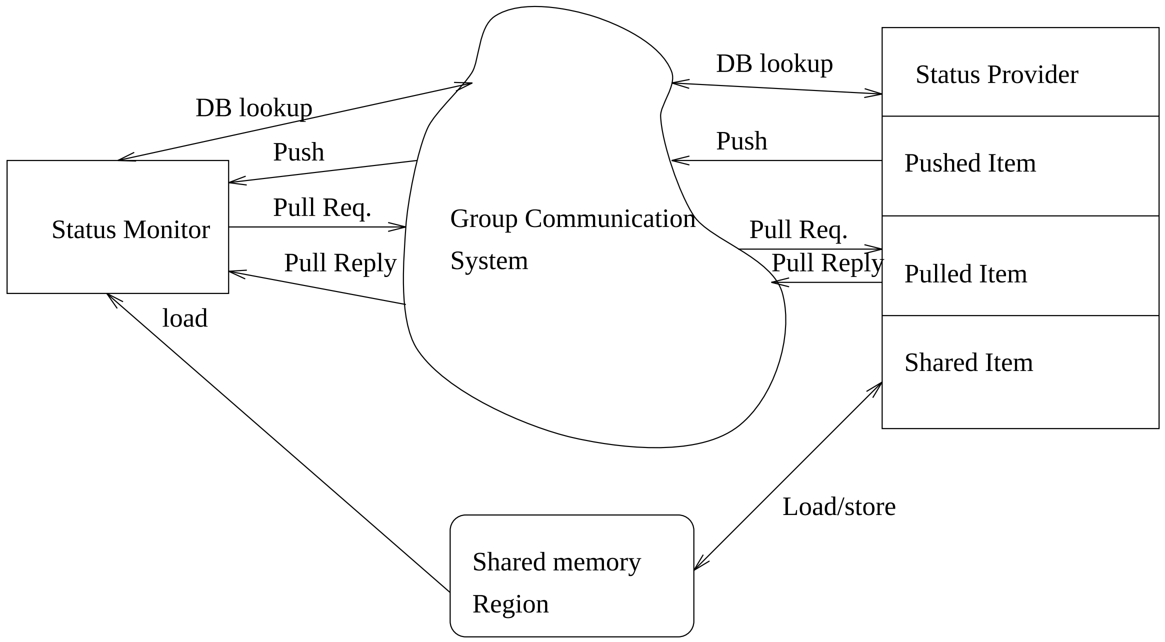

Figure 16: Status Path Simplified through group communication

most recent status value for pushed items to which the monitor has connected.

The figure shows a single system configuration. We have demonstrated how shared memory status items can be transparently mapped to pulled status items via our status factory. We have also demonstrated that status monitors can transparently recover from failures in the provider when the provider is restarted. We have also demonstrated that this recovery can work even in the presence of status item migration (e.g. to another system), or a change in the transport mechanisms from incarnation to incarnation of the status item.

7 Conclusions and Future Work

We have posed three questions at the start of this article:

1. Can object oriented software techniques make it easier to write data acquisition systems.
2. Does distributed and persistent object technology provide useful tools to deal with database consistency problems.
3. Can distributed objects or collective communication help resolve command targeting problems.

Our prototyping work shows that, through object oriented development, it is possible to reduce the programming burden on the user programming a specialized component of a data acquisition system. This can be done in several ways.

First, careful design of object classes can hide differences which the user should not have to care about. This was demonstrated in our class hierarchies for status distribution and monitoring. Further bundling database manipulations into class constructors, or factory methods hides even the fact of difference from the user.

Second, by producing a good application framework, it is possible to reduce the volume of code that a user must write to integrate a program into an existing system. Since the application framework represents the re-use of a debugged design of a generic DAQ application, integration of user programs is much safer than if the user had to re-invent the entire application infrastructure.

The key to a good application framework is careful incompleteness, allowing the user a large number of safe hooks in to the application main loop to deal with special circumstances. Once the application framework exists, the number of lines of code the user must write to produce a data acquisition program is greatly reduced.

On the negative side, however, effective use of an application framework requires the mastery of a potentially large class library. Furthermore, while increasingly students begin their work at laboratories with a cursory knowledge of object oriented software development, it will be some time before this knowledge is universal amongst laboratory users. Education will be necessary if use of these powerful techniques is to be successful. In addition it will be necessary for data acquisition systems targeted for use in the near future to provide non-object oriented entry points to the system. That this can be done is demonstrated in [Nol94].

Object oriented technologies like persistent storage and distributed object managers are strangely not the help we hoped they might be. CORBA usage was somewhat disappointing While useful in point to point communication it is, with the exception of one vendor's product, not useful in resolve the collective communication and database issues that we need to solve. The implementation of ILU, the CORBA-like DOM manager we used, is a warning about the dangers of inflexible application frameworks. Sufficient flexibility must be provided in the framework's main loop to allow software to do tasks not anticipated by the system designer.

Persistent object systems are a bit more helpful. They can resolve the issue of maintaining persistent system state which is not provided by any single process. We also hope that these systems can be used to smooth the transition from experiment to analysis. Unfortunately, the technology in this area is still immature for a production system. However it is probably appropriate to do some larger scale prototyping with the understanding that standards will eventually come along.

Our most pleasant surprise was the ability of group communication to solve the consistency problem. It was clear that group communication would deal with the targeting and process database problem, since targeting and process databases are essentially what a group communications system is made of.

Our protocol for performing distributed database look ups via group communication is reminiscent of IP's use of the Address Resolution Protocol (ARP) to map IP numbers to physical addresses. The match is exact not only in what these protocols accomplish, but also how they accomplish their goals.

We feel in retrospect that it has been a mistake to separate metainformation about services provided by active agents of the DAQ system from those agents. It is this separation that has caused the consistency problem in the first place. Combining group communication and the information and behavior hiding aspects of object oriented languages allow us to maintain internal meta-data in a safe and transparent manner to user level code.

This study was done in the context of a DAQ system reference model which is a close match to the UNIDAQ system. The authors propose to develop a DAQ system, incorporating group communications and using object oriented development techniques, based on the simplified reference model described in VI. Key to the rapid implementation of this system will be a more detailed definition, design and implementation of an application framework or small set of application frameworks which will support the development and integration of processes within the reference model.

Many practical problems will have to be solved to achieve this goal. We invite others in our field to collaborate with us in this exciting development effort.

8 References

References

[Al94] The Obst Tutorial J. Alt et al. 1994 Forschungszentrum fur Informatik Karlsruhe FZI.049.2

[Ap95] What is OpenDoc? Apple Computer Inc. online document at *http://dev.info.apple.com/du/intro_to_opendoc/iod0_index.html* 1995

[Ap96] Apple Developer World Apple Computer *http://www.devtools.apple.com* 1996

[Ar9] *Object Oriented Systems in High Energy Physics* C. Arnault *Computing in High Energy Physics 1992* pg. 75

[Arl95] Ode 3.X User Manual R. Arlein et al. AT&T Bell Laboratories Murray Hill Also available online: *http://www.nada.kth.se/ sam/Pim/Db/userguide.html*

[Arn96] The Java(tm) Programming Language K. Arnold, J. Gosling Addison Wesley 1996

[Ba93] UNIDAQ Documentation Set V2.2, Software for UNIX-based Data Acquisition R. Ball (ed.) SDC Technical Report: *SDC-93-573*, University of Michigan Technical Report *UM-HE-93-29* (1993)

[Ba95] *Communications and Control Mechanisms int the Unix-Based UNIDAQ Data Acquisition Software* R. Ball et al. Procedings of the 1995 IEEE Conference on Real-Time Computer Applications in Nuclear, Particle and Plasma Physics East Lansing, 1995 pg. 152

[Bar96] *Fibre Channel Based Network for the Euroball Experiment* G. Barazza et al. abstract submitted to the DAQ96 Networked Data Acquisition workshop Scheduled for November 1996 in Osaka Japan.

[Be94] *Data Flow Manager for DART* D. Berg et al. Computing in High Energy Physics 1994 pg. 453

[Bi94] *Performance of the ISIS Distributed Computing Toolkit* K. Birman, T. Clark Cornell Computer Science Department Technical Report *TR94-1432* online: *http://cs-tr.cs.cornell.edu/TR/CORNELLCS:TR94-1432*

[Bj88] *ADVANCE: An Object Management System* A. Bjornerstedt, S. Britts Procedings of OOPSLA88

[Bo91] Object Oriented Analysis and Design with Applications G. Booch Benjamin Cummings 1991

[Br94] *Experience Building Object-Oriented Systems* A. Breakstone et al. Computing in High Energy Physics 1994 pg. 41

[Ca90] *The Exodus Extensible Database Project: An Overview* M.J. Carry et al. University of Wisconsin Technical Report *TR-WISC-CS-TR 808.*

[Co91a] Object Oriented Design P. Coad and E. Yourdon Prentice Hall 1991

[Co91b] Object Oriented Analysis P. Coad, and E. Yourdon Prentice Hall 1991

[Do96] *The Transis Approach to High Availability Cluster Communication* D. Dolev, D. Malki Communications of the ACM, 39,4 April 1996

[Fe90] A storage System for Object Oriented Applications M.F. Fernandez Brown University Technical Report *LS-90-27*

[Fl92] X Toolkit Intrinsics Reference Manual D. Flanagan (ed.) O'Reilly and Associates 1992

[Fo85] *Progress on the Data Acquisition System at NSCL* R. Fox et al. IEEE Transactions on Nuclear Science NS-32 No. 4 (1985)

[Fo93] *Portable Dataflow In Unix* R. Fox et al. IEEE Transactions on Nuclear Science Vol. 41 No. 1 1994 pg. 161

[Fo95] *The Xamine Online/Offline Display Program* R. Fox et al. IEEE Transactions on Nuclear Science Vol. 43 No. 1 Pg. 55

[Ga95] Design Patterns: Elements of Reusable Object-Oriented Software E. Gamma et al. Addison Wesley 1995

[Ge94] PVM3 User's Guide and Reference Manual A. Geist et al. Oak Ridge National Laboratories 1994 publication *ORNL/TM-12187*

[Go89] Smalltalk-80: The Language A. Goldberg, D. Robson Addison Wesley 1989

[Gos86] The Java Language Environment a White Paper J. Gosling, H. McGilton *http://java.sun.com/doc/language_environment/*

[He94] *The CEBAF Online Data Acquisition System* G. Heyes et al. Computing in High Energy Physics 1994 pg. 122

[Hi95] The Magic Carpet for Network Computing: HORB Flyer's Guide S. Hirano 1995 online document e.g. *http://ring.etl.go.jp/openlab/horb/doc/guide/guide.htm*

[Io95] An Introduction to Orbix(tm)+ISIS Iona Technologies Ltd and Isis Distributed Systems, Inc. online document at *http://www.usa.iona.com/Orbix/isis/Summary.html*

[Ja92] ConceptBase V3.1 user manual M. Jarke (ed.) Technical Report Aachener Informatik-Berichte *92-17*.

[Ja95] *ConceptBase - a deductive object base for meta data management* M. Jarke, R. Gallersdorfer, M.A. Jeusfeld, M. Staudt, and S. Eherer Journal of Intelligent Information Systems, Special Issue on Advances in Deductive Object-Oriented Databases Vol 4. No 2, 167-192 1995

[Jan96] ILU 2.0 alpha 8 Refrence Manual W. Janssen, M. Spreitzer Xerox Palo Alto Research Center May 1996

[Ki90] *Architecture of the Orion Next-Generation Database System* W. Kim et al. IEEE Transactions on Knowledge Data Engineering March 1990

[Kr96] Java API Overview D. Kramer Online document at: *http://java.sun.com/products/apiOverview.html*

[Lu96] Programming Python M. Lutz O'Reilly and Associates 1996

[Mi94] *The Zeus Message Passing System* M. Milewski et al. Computing in High Energy Physics 1994 pg 177

[Mo91] The CLOSQL Query Language S. Monk Computing Dept. Lancaster University, Lancaster Technical Report No.*SE-91-15*

[MP94] The Message Passing Interface Standard Message Passing Interface Forum 1994 online at *http://www.mcs.anl.gov/mpi/mpi-report/mpi-report.html*

[Mu92] *A Transaction Model for an Open Publication Environment* P. Muth et al. ul Data Base Transaction Models for Advanced Applications A.K. Elmagarid (ed.) Morgan Kaufmann 1992

[No] Nova buffer manager for SDC portable DAQ Users manual M. Nomachi, Y. Takeuchi part of UNIDAQ distribution kit available from *http://kekonline.kek.jp*

[No94] *UNIDAQ* M. Nomachi et al. *Computing in High Energy Physics* 1994 pg 114.

[Nol94] *FARFALLA: C++ Data Management That Even FORTRAN People Can Love* B. Nolt and C. Walter. Procedings of Computing in High Energy Physics 1994 page 271.

[Ny92] X Toolkit Intrinsics Programming Manual A. Nye and T. O'Reilly O'Reilly and Associates 1992

[Odmg] Object DAtabase Management Group: *http://www.odmg.org*

[Oh89] *Database Programming in Machiavelli – a Polymorphic Language with Static Type Checking* A. Ohori, P. Buneman, V. Breazu-Tannen Procedings of SIGMOD 1989

[Ol93] *DART - Data Acquisition for the Next Generation of Fermilab Fixed Target Experiments* G. Oleynik et al. IEEE Transactions on Nuclear Science Vol 41 No. 1 Pg. 45

[Ol95] *Fermilab DART Run Control* G. Oleynik et al. ul IEEE Transactions on Nuclear Science Vol 43 No 1 pg 20

[Omg95] The Common Object Request Borker: Architecture and Specification the Object Management Group online: *http://www.omg.org/corba2/cover.htm*

[Oofaq] *http://www.geodesic.com/oofaq/out-toc.html*

[Py96] Python web pages are located at *http://www.python.org*

[Re94] Design and Performance of HORUS: A Lightweight Group Communications System R. van Rennesse, T. Hickey, K. Birman online Cornell Technical Report: *http://www.cs.cornell.edu/Tnfo/People/rvr/papers/arch/arch.html*

[Ru91] Object Oriented Modeling and Design W. Rumbaugh et al. Prentice Hall 1991

[Sc94] User's Guide Obst Release 3-4 B. Scheifer et al. Forschungszentrum fur Informatik Karlsruhe *FZI.032.7*

[Sh88] Object-Oriented Systems Analysis: Modelling the World in Data, S. Shlaer and S. Mellor Prentice Hall 1988

[Sh92] Object Lifecycles: Modelling the World in States S. Shlaer and S. Mellor Prentice Hall 1992

[St91] The C++ Programming Language B. Stroustrup Addison Wesley (2nd edition) 1991

[Tr92] *Meta Object Management and its application to the Datbase Revolution* M. Tresch Proc 11'th International Conference on the Entity-Relationship Approach 1992

[Uh94] The Object Management System of Stone Release 3.4.3 J. Uhl Forschungszentrum fur Informatik Karlsruhe *FZI.027.5*

[Wa93] *CODA: A Scalable, Distributed Data Acquisition System* W.A. Watson III et al. IEEE Transactions on Nuclear Science vol 41 No. 1 pg 61

[Wh85] *VAX Online System – A Modular Approach* V. White, P. Constanta-Fanourakis, P. Heinicke, D. Petravick, L. Quigg IEEE Transactions on Nuclear Science NS-32 No. 4 pg. 1384