Andrey D. Petrov
Accelerator Controls Department
Accelerator Division
630.840.6877 (phone)
630.840.3093 (fax)
apetrov@fnal.gov

June 19, 2007
Revised August 10, 2007

New Hybrid Application Environment for Controls

Abstract

This technical note describes the idea of a new hybrid application environment for the data acquisition and control systems at Fermilab, defines a list of specific issues the application developers are currently facing, and sets forth general requirements for the new system.

Reasons for a New Application Environment

With the introduction of two alternative control systems at ILC Test Facilities a bunch of new technical stuff comes on the scene. The user applications, protocols, and API are not novel in fact, but certainly different from those we have had before. Let's skip the things on the lower level of these systems—front-end architectures and transport protocols—and look at the façade.

Each product already includes a set of specific applications, capable to perform general functions. It's important that the users can start using a control system right away, with the provided tools, and don't need to spend time and efforts on in-house development.

In the course of time, however, more and more custom applications will be implemented. Once installed, open software systems start expanding—this fact is inevitable. For a working product, there is always a need of integration with other systems, new demands from its users, and new challenges from the outside environment—such as network security issues—that have to be addressed. Given the number of computer technologies around, every conceivable gadget will be ultimately written: a web-service provider (because non-HTTP ports are blocked by the firewall), a bridge to ACNET, new server- and client-side applications, nightly reports, AJAX-based displays with SVG images, and everything else not included in the initial design. It happens naturally to fill the gap between what the core system provides, and what's really needed and can be implemented with the means at hand.

Most of the user applications in ACNET, DOOCS, and EPICS can run, as a matter of fact, only on designated hosts. An ordinary user can't launch them on his or her own PC, and has to utilize a remote access tool, such as X terminal or VNC. In some cases, this mode of operation is inherited from a legacy system, in others it has to be followed due to security restrictions. As almost everything runs in a central location, the systems are not designed to distribute data to remote clients. It's impossible, in principle, to write a data acquisition program (within the current frameworks) that would run outside certain limited area. Other pitfalls include the need of high bandwidth to transfer graphics, and the risk of typing passwords over insecure connections. However, despite the drawbacks, the remote mode of operation is widely adopted in controls. There are two main reasons for that: Firstly, it provides a fairly simple manageable environment in which all the users work. Secondly, the application developers don't have to worry about various network and security issues.

A manageable software environment is characterized by, at least, four features:

- All nodes have similar configuration. All required components are preinstalled. The configuration can be easily restored if the system crashes.
- There is a well-defined way of distributing applications.
- The system runtime parameters, such as a list of users logged in, a list of running processes and threads, CPU and memory loads, contents of log files, can be monitored without exertion.
- The code developed on-site is subject to version control. All components included in the production release are consistent with each other.

Application frameworks offered by the existing control systems were designed many years ago and currently are seemed to be too restrictive and somewhat obsolete. It doesn't depreciate the value of existing applications. However, instead of developing a new C++ program that would run via X terminal, the users may want to obtain and process data locally, or through a web-based GUI. These and others technical solutions were not foreseen in the original system designs, thus not supported by the existing application infrastructures. As a result, around the core systems emerges a set of miscellaneous components and protocols, which live on their own. This ranges from a reasonably structured Tomcat web server to a bunch of individual scripts that may or may not provide log files and diagnostics. They all have to be somehow deployed, started when needed, and otherwise maintained.

For example, let's consider the ACNET Console (*http://www-bd.fnal.gov/controls/public/linux-console.html*). This tiny applet provides read-only access to legacy ACNET applications. Behind the scene, the applet (which runs on the client) connects to another server-side application that dispatches all the traffic. That server module doesn't fit any existing application framework, so besides the main business functions it has to:

1. Implement a custom application-level protocol on top of TCP/IP.
2. Include a trivial web server to show up the status and internal diagnostics.
3. Deal with log files.

In order to run, it needs:

4. A custom script that uses proper version of Java and application libraries, and changes the effective UID.

In order to deploy the module, the system administrator has to:

5. Create two directory with different permissions, and deploy the recent version of scripts and libraries.
6. Configure the module to be reloaded if the system restarts.

Besides obvious difficulties with maintenance, the development of such new applications requires unneeded work on services that ought to be provided by an external environment. This includes popular transport protocols, security, logging, diagnostics, and a set of central services.

In a practical perspective, an application developer has two choices. He or she can either keep on writing programs inside the old traditional frameworks and abandon more modern technological features; or one can start doing something new, but assume the full burden of deployment, maintenance, and other extra work.

Current Issues

There are several general issues with the application environments that are persistent, to various degrees, in the present control systems:

Availability of Data, Applications

In the existing systems, a legitimate user can't start a functional application where needed, because either the application can't be installed or can't run on that host, or data from the control system can't reach it. The data is not limited by only device readings and settings, but also includes database connections and such. This problem is currently solved with third-party remote access tools (X terminal, VNC, Windows Remote Desktop, VPN), which add an extra level of complexity. Also, some users may want to use applications locally (application developers are among them), whereas others prefer to use on remote preconfigured hosts (occasional users). Both options must be honored.

The application environment must support redundancy, so it can provide required services even if the part of the infrastructure is down. All consumers should be able to get data transparently from more than one data provider. In general, a consumer doesn't need to know which provider it's connected to.

Security

The lack of availability is mostly caused by the lack of adequate security. When the application environment can't control access and protect channels in the way that complies with a security policy, this is done by the firewalls, which simply cut the traffic off. Ideally, a control system should reuse security credentials provided externally (e.g., Kerberos tickets), and authenticate clients on the transport (not application) level.

Common Application Context

All components of the system—event though they can be physically distributed—must run in a common context, which provides all internal “plumbing and wiring”. This includes links between components (both local and remote), central management (deployment, starting, and stopping), monitoring (list of running entities, use of resources, contents of log files, internal diagnostics), and others. The application context's architecture should promote modularity of components and reuse of services.

Application Development

Programming users must have a reasonably simple way to develop new applications and incorporate them into the system. For multi-tier systems, there must be a transparent local development process for all types of components. The infrastructure should enforce version control and consistency of production builds. There should be be a recommended configuration for the development environment (also installed on shared servers), and an option for those who prefer to use an arbitrary IDE.

Integration of Different Control Systems

Traditionally for the controls, application environments have been offsprings of the corresponding low-level data acquisition protocols. They are not compatible with each other, as every system has its own communication protocol to the clients, different data acquisition (DAQ) API, and separate resource namespaces. With the introduction of multi-tier architectures it's shouldn't be the case. The middleware can solve the issues of integration by providing two functions. Firstly, it can implement a central service that makes the data from one low-level data acquisition systems available for others. This would allow to reuse existing tools (user application, central cervices on a lower level) to process foreign data. Secondly, it can provide a new common DAQ API for the clients, along with an up-to-date connection protocol that works in a reliable and secure manner.

The Idea

The major issues with applications can't be solved within frameworks of the existing control systems. Conceptually, low-level DAQ systems should be separated from the top-level application infrastructure. Both parts are equally important, but deal with different concerns. In early times, the controls application environment had to be custom to a large degree, because of lack of readily available solutions. Now the relevant technologies are quite elaborated and widely used, so it doesn't make sense to stick with in-house software.

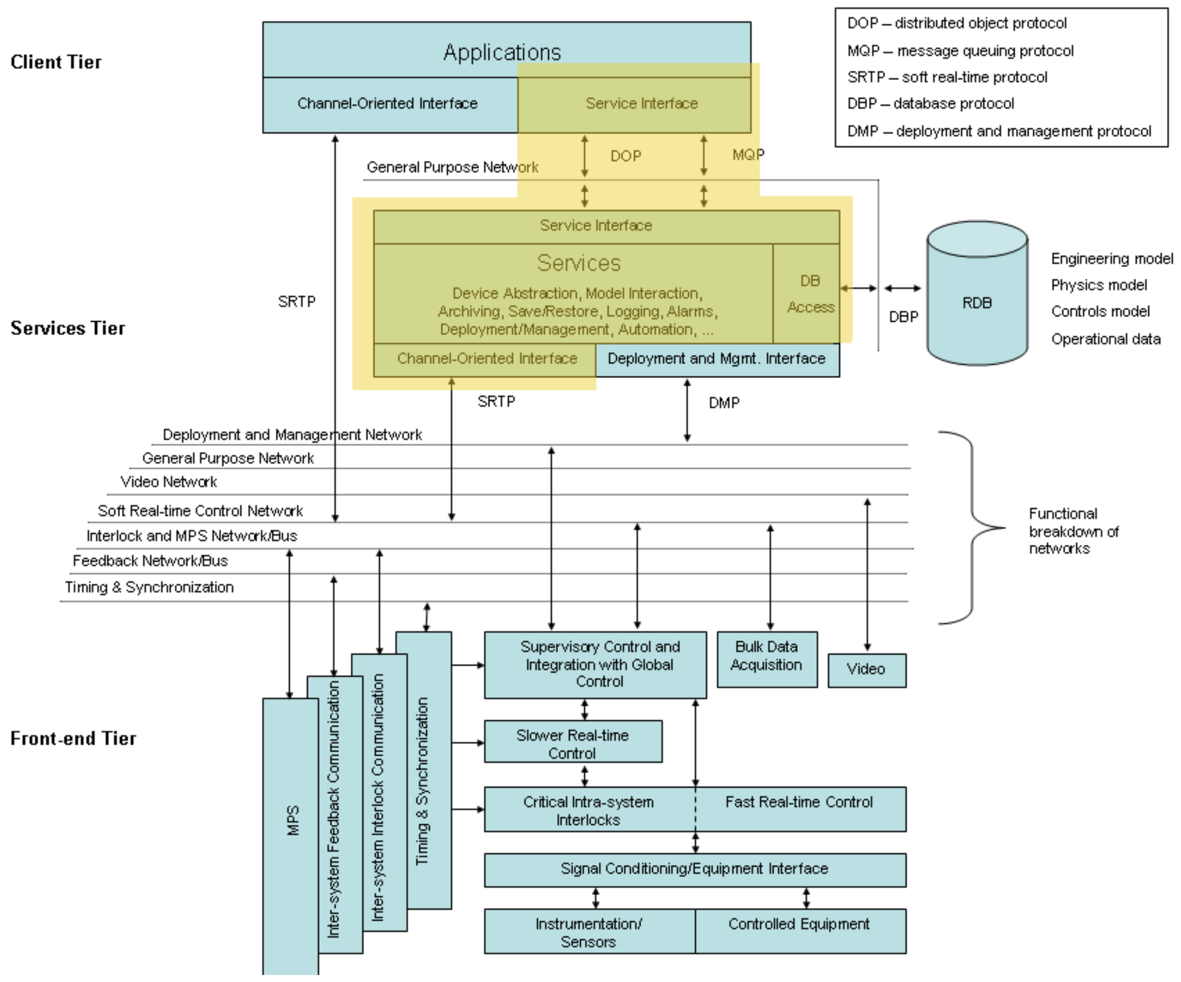
It is proposed to start the development of a new distributed hybrid application infrastructure, independent of any particular data acquisition protocol, with the following two objectives:

- **In a short term,** it will benefit the existing control systems—ACNET, EPICS, and DOOCS—by providing a flexible operation environment and a set of central services that are not currently available or need to be improved. New extensions for the core systems will run inside that common environment. The common infrastructure will also contribute to the integration between the different low-level data acquisition systems, by providing a new unified client-side DAQ API, common resource name space, and bridges for cross-availability of data. The existing applications will be able to run unchanged.
- **In a long term,** it will be employed for testing and demonstration of candidate technologies relevant to the R&D Program for ILC Controls. There is a number of different products on the market. Right now we don't have enough practical experience whether and how they can be applied to build an application infrastructure for the accelerator controls. Even if a system performs fairly general functions, it is always designed with a particular group of customers in mind. The business of data acquisition and controls is quite unique and has its own set of requirements—although some of the

features are seemed to be common with more traditional sectors, such as telecommunications, e-commerce, or stock market. As the ILC Controls R&D is focused on high-availability, for the application environment it translates to such essential features, as monitoring and management, redundancy, and conflict avoidance.

Scope and General Requirements

In [1] the authors suggested to use a multi-tier architecture for the future ILC control system. The hybrid application infrastructure proposed in this document fits this model as shown on the diagram below.



Client Tier
Applications
Channel-Oriented Interface
Service Interface
DOP
MQP
General Purpose Network
DOP – distributed object protocol
MQP – message queuing protocol
SRTP – soft real-time protocol
DBP – database protocol
DMP – deployment and management protocol
Service Interface
Services
Device Abstraction, Model Interaction, Archiving, Save/Restore, Logging, Alarms, Deployment/Management, Automation, ...
DB Access
DBP
RDB
Engineering model
Physics model
Controls model
Operational data
SRTP
Services Tier
Channel-Oriented Interface
Deployment and Mgmt. Interface
SRTP
DMP
Deployment and Management Network
General Purpose Network
Video Network
Soft Real-time Control Network
Interlock and MPS Network/Bus
Feedback Network/Bus
Timing & Synchronization
Functional breakdown of networks
Front-end Tier
MPS
Inter-system Feedback Communication
Inter-system Interlock Communication
Timing & Synchronization
Supervisory Control and Integration with Global Control
Bulk Data Acquisition
Video
Slower Real-time Control
Critical Intra-system Interlocks
Fast Real-time Control
Signal Conditioning/Equipment Interface
Instrumentation/ Sensors
Controlled Equipment

The service tier of the new application environment will include:

- An application server that runs a set of loosely coupled components, providing an adequate "plumbing and wiring" between them. That includes common configuration, logging, and monitoring functions. Each components (or group of components) is responsible for an individual service. The candidate services are:

    - Data acquisition service, which makes the data from the front-end tier available to the clients. Also, it can act as a bridge between different low-level control systems. The data acquisition service itself consists of several smaller service, providing data consolidation, front-ends *frontending*, conflict avoidance, and high-level data acquisition logics.
    - Security service (authentication & authorization, user database).
    - Naming and directory service (device database).
    - A simple service to run shell scripts.
- Application index, responsible for distribution of user applications.
- Three pluggable adapters for the individual low-level control systems.
- Remote interfaces to the clients and siblings. Because of different kinds of user applications, at least two distinct protocols will be supported: a web-based protocol (such as SOAP, XML-RPC, etc), and a binary RPC or messaging protocol (similar to CORBA, Java RMI, etc).

The client tier will include:

- Basic device and data abstractions.
- General data acquisition API that supports both synchronous (blocking) data acquisition and subscriptions with asynchronous callbacks.
- Corresponding remote interfaces to the service tier.

In this perspective, the existing control systems are put inside the front-end tier. From the point of view of an individual data acquisition system, the whole application infrastructure described here is no more than just an ordinary user application.

Various technical considerations:

1. The system shall use reasonably standard technologies and off-the-self open-source products whenever possible. Whereas individual components implementing specific services have to be custom, the larger things, such as the application server, must be generic.
2. It is assumed that the core of the middle tier will be developed on Java. Yet, all the transport protocols must be language-independent, so the clients can use other languages when needed. Multiple languages should be promoted. As the Java platform now supports the direct use of other languages, the middleware itself can be made hybrid.
3. All source code (in all languages) must be properly organized and [strictly!] the subject to version control. In particular, we should carefully track the issues of licenses on third-party products, and on the own code. The central building system must be able to make a complete production release and deploy it accordingly.
4. The number of dependencies inside the code shall be minimized. The client-side applications must be realistic about available resources (total size of the executable, memory footprint, network bandwidth).

5. Security features (access control, TLS) are built in the system from the day one.

References

[1] Saunders C., et al., “Control System Architecture Model for ILC Reference Design” ILC-DOC-...