# Data Acquisition System Comparison for ESS

**October 2012**

**Mark Könnecke**
**Paul Scherrer Institut**
**Switzerland**

## Introduction

In this document various current data acquisition systems used at neutron and synchrotron facilities are compared against each other. As this survey is aimed at designing a data acquisition system for ESS, special attention will be given to aspects of neutron data acquisition.

The term data acquisition (DAQ) system is ambiguous. For the purpose of this document it means the complete software system: neutron or photon detection and capture, slow controls such as motors, sample environment and such and the coordination of all these components. It also includes the user interface used for running an instrument.

For each system the following topics will be covered:

- An overview of the system
- Hardware and software requirements for the system
- How the system accesses hardware
- Neutron data acquisition implementation of the system when appropriate
- Implementation of higher level functionality and sequencers when appropriate
- User interfaces
- Data handling
- Features of the system
- Management issues like
  - Support
  - Human resources requirements of the system

Another clarification: in this document the word instrument is used. To the author an instrument is a complete beam line from source to sample and detectors.

There is also a short glossary of computing terms at the end of the document.

## TANGO

TANGO is a distributed data acquisition and control system which has been developed in a collaboration between ESRF and seven other partners including SOLEIL, ELETTRA, ALBA, FRMII and Maxlab. Some institutes like ILL are users of Tango. Functionality in Tango is implemented in TANGO device servers. Communication between TANGO device servers and clients happens via CORBA and ZeroMQ. What is used on top of TANGO in order to implement higher level functionality varies:

- SPEC is still used at the ESRF beam lines as a sequencer
- Many beam lines use scripts in a variety of scripting languages to run their instruments
- ALBA has developed a sequencer called Sardana on top of TANGO which starts being used at other facilities too.

### *TANGO System Requirements*

TANGO runs on all popular operating systems: Linux, Windows and Macintosh. However, most usage is on Linux. Besides the OS, TANGO requires the following software:

- OmniORB
- Jacob
- gcc 4.+
- Java
- Python

All these requirements are free software.

In terms of network protocols TANGO uses CORBA and ZeroMQ for events (since Tango V8).

### *TANGO Hardware Access*

Hardware access (and higher level functionality) is implemented in TANGO device servers. A TANGO device server is a full CORBA server. Each device server can hold multiple devices. And any given device can implement a number of commands, attributes and properties. There exist device abstract classes which describe which commands, methods and properties to expect for devices of that class. For example the device abstract class for motors would detail which properties and functionality to expect for a motor. The abstract classes are not pure abstract - they can implement common behavior if needed.

TANGO facilitates the development of new device servers through several techniques. On the programming language level TANGO provides interfaces:

- for configuring the properties and functions available on the device
- for the implementation of the properties and functions of the device
- a state machine for every device class means every device has state and state transitions

Programming device servers thus comes down to inheriting from base classes, extending them and implementing a state machine

Moreover there is a code generator called POGO. POGO takes as input a text file which contains a description of the properties and functions of a device in a domain specific language. POGO then creates a skeleton TANGO device server for you. A programmer just has to fill in the code.

All TANGO software is developed against the TANGO library. The TANGO library provides wrappers and implementations for common system services like network protocol, threads, polling, logging, configuration, bus access etc. The advantage of this approach is that whenever any of the underlying services changes, then only the library needs to be modified. Device servers stay unaffected and just link against the new library. This helps ensuring long term maintainability of the software.

TANGO device servers can be implemented in any the programming languages:

- C++
- Java
- Python

C++ is however the main development language. Due to the shared network protocol, device servers implemented in any of these programming languages can be freely intermixed.

Clients can interact with TANGO device servers RPC style through CORBA. Clients can also subscribe to TANGO device servers in order to receive automatic notifications about device state changes via ZeroMQ.

There are plans to replace CORBA through ZeroMQ completely in the future.

## *Neutron Data Acquisition*

TANGO is mainly used at synchrotron radiation facilities. It thus has efficient support for CCD cameras but so far not for neutron data acquisition. But there is nothing in TANGO which prevents its use for neutron DAQ. Tango is used at two neutron facilities - FRMII and ILL. In contrast to EPICS, TANGO has no problems with transferring arrays. Large arrays are transferred at network speed.

## *Implementation of Higher Level Functionality*

TANGO offers several choices for implementing higher level functionality:

- Higher level functionality can be implemented in specialized TANGO device servers. The TANGO device server model is flexible enough to implement almost anything.
- Another option is to implement higher level functionality in special client programs. Such programs can be implemented in any language with a CORBA interface. There also exist TANGO bindings for many programming and scripting languages(Python, Matlab, IgorPro, Labview). Bindings can be written for any language which can call C, C++, Java or Python code
- Tango devices can be organized hierarchically to hide and better manage complexity in systems
- One of the TANGO aware sequencers can be used. Currently these are SPEC and Sardana.

SPEC is a simple scripting language containing primitives for accessing DAQ hardware and more importantly a macro system which allows to define functionality as SPEC macros. SPEC is used at many synchrotron facilities. However, SPEC is very old, commercial and maintained by one person who has already reached retirement age. Thus I would not recommend SPEC for any new project like the ESS.

Sardana adds the following elements to TANGO:

**A Device Pool**

A central container which holds all devices and pseudo devices belonging to an instrument. The device pool also contains interfaces to common hardware. This makes it possible to directly access devices by writing controller classes, thereby bypassing the TANGOs device server model. The device pool server can be queried to locate devices of specified types or names.

**A Macro Server**

A server which allows to execute macros written in python against the device pool. Macros can be dynamically loaded and reloaded.

**A CLI client**

A command line client with a SPEC look alike syntax which interacts with the macro server to execute operations against the DAQ system

**A GUI**

A graphical user interface which allows to view the device pool and macro server. Pythons introspection facilities are used to document available macros to the user. It is also possible to match Macros with appropriate UI classes in order to arrive at a tailor made UI component. The whole GUI can be configured to a wide range of applications.

Sardana also adds additional functionality to TANGO. The most important are:

- A simulation mode
- NeXus data file writing
- Extension points to add instrument specific functionality
- A GUI toolkit for creating instrument specific user interfaces

## *User Interfaces*

Besides Sardana, there is no single, general purpose user interface which sits on top of TANGO. The ATK toolkit written in Java/Swing is used for most general purpose tools. Each institute has different competences and therefore has preferred to use graphical tool kits of their choice. The most common ones are Java/Swing and C++ or Python QT. There are also many specialized graphical user interfaces. The TANGO distribution also contains several graphical tools for managing a TANGO system. But these tools are usually not something an external user is meant to interact with.

Scripting is implemented through TANGO adapters for a variety of scripting languages.

## *Control System Features*

**Configuration**

TANGO Device servers configure themselves from a central database.

**Addressing**

Parameters in TANGO device servers are accessed via hierarchical path strings like in a unix file system

**Simulation mode**

TANGO by itself does not support a simulation mode. But Sardana does.

**Logging**

There are configurable logging options. Logs can be switched on and off and can happen to both local files and to a central service on the network.

**Debugging**

There is no special debugging support in TANGO.

**Error propagation**

Happens via CORBA exceptions

**Remote access**

Is done via remote desktop solutions like NX. There is also the option to tunnel TANGO CORBA calls via the WWW or JBus. An HTTP to Tango bridge is under development.

**Integration of sample environment and user hardware**

Either through a TANGO device server or by scripting from client programs by direct bus access. There is also a standard solution for digital and analog I/O with user hardware. Tango device servers can be embedded into device hardware (running embedded Linux for example) and be configured from a file.

**Security**

Some security by restricting access to certain networks and users. Protects against mistakes, not serious hacking.

**State Change Propagation**

Every device has state. Clients are informed of changes by subscribing to STATE_CHANGE events

**Unit Testing**

50 % coverage and increasing for TANGO library code.

**Parallel Processing**

TANGO device servers work in parallel, thus parallel processing is supported. Data protection happens via mutexes at device server level. Clients can either poll for state changes or subscribe to events. Every client connection has its own thread.

## *Management Issues*

This is an estimate of the learning curves and support personnel required to run TANGO.

| *Activity* | *Time required* |
|---|---|
| Teaching a user | 2-3 h |
| Teach an instrument scientist | 1 months |
| Teach device development | 3 days |
| Induce new IT staff | 6 months |
| Develop drivers | days to weeks |

ESRF currently employs 17 staff plus community support to run 42 beam lines. ESRF beam lines are more complex then neutron ones: a ESRF beam line can include up to 100 motors, 20 monitors, 2 1D detectors and 5 2D detectors.

TANGO is used at several synchrotron facilities. Thus there is a good level of support and an ample stock of existing device drivers and modules which can be reused.

# NOMAD

NOMAD is a data acquisition system used and developed at ILL. It is a client server data acquisition system consisting of a central server and a matching graphical user interface. The UI and the server communicate with each other through CORBA. The NOMAD server is written in C++. The graphical user interface is written in Java using the SWT toolkit but not eclipse-RCP.

## *System Requirements*

NOMAD runs on Linux 32 bit. A port to Windows has been done. As CORBA object request broker, OmniORB is used.

## *Hardware Access*

All hardware access happens through the NOMAD server. HW access in NOMAD is organized in layers:

1. At the bottom is a bus layer which abstracts the communication through a variety of buses like VME, TCP/IP, RS-232 etc.
2. On top of the bus layer is the driver layer.
3. On top of the driver layer sits an abstraction layer which allows to configure the format of the commands to send and the replies to parse in a XML file.

However, when a driver cannot be realized through configuration, a new driver needs to be written in C++ and linked into the NOMAD server. In this process the developer is supported by the NOMAD server framework which provides not only for bus access but also for threading and other primitives.

## *Neutron Data Acquisition*

The ILL uses PowerPC VME onboard computers for neutron data acquisition. The onboard computer receives neutron events through mezzanine I/O modules. The PowerPC computer does not run an operating system but rather a selectable histogramming process directly on the hardware. The normal mode of operation is histogramming. After the end of data acquisition the resulting histogrammed data is transferred via an VME interface card to the data acquisition computer at 40MB/second. With more advanced hardware this can increase to 3GB/sec. This neutron DAQ system is capable to process neutrons at 5MHZ.

There is also an event DAQ mode. In this mode two buffers exist in memory which get filled alternately. Finished buffers are read by the DAQ computer. This mode of operation is capable of 1TB/day.

## *Implementation of Higher Level Functionality*

All higher level functionality is implemented in the NOMAD server. Scans and scientific controllers are embedded. Scientific controllers translate physical values, like wavelength, into movements of devices. C++ coding, compiling and linking is required to add functionality to NOMAD. There is the intention to simplify this a little by dynamic loading.

A central data structure in the NOMAD server is the command queue. More complex user commands are broken down into simpler commands which are queued in the command queue. A scheduler then takes care of executing the command queue. A separate scheduler and queue exist in order to support a simulation mode.

NOMAD supports different data file formats for different user groups. Increasingly NeXus files are written through a template system.

## *User Interfaces*

The graphical user interface to NOMAD is the NOMAD GUI. The key view of the NOMAD GUI is a tree view of the instrument. This tree view is filtered according to the role of the user: user, instrument scientist and support staff. Different user roles see different levels of detail of the instrument. Through this tree view the instrument can be run or batch files can be created graphically. The NOMAD GUI supports loop constructs within the graphical batch editor. A separate window is used to display the data currently being collected in the instrument. The NOMAD GUI is implemented in Java using the SWT UI toolkit. GUI components are defined in XML files; thus it is possible to change the look and function of UI components without recompilation.

The NOMAD GUI has been ported to Android.

The NOMAD GUI communicates with the NOMAD server via CORBA. Updates of the instrument status are propagated to the NOMAD GUI through CORBA notifications.

A scripting interpreter is included in the NOMAD server. NOMAD's scripting language is NOMAD specific. A parser was added to understand the syntax of the MAD software, the predecessor of NOMAD. All new structures such as loops and control constructs are only available in NOMAD's own scripting language.

## *Control System Features*

**Configuration**

Via XML files managed by the NOMAD server. The configuration files represent the total instrument with all hardware and sample environment options. Devices can be disabled/enabled according to the needs and use cases of the instrument. The configuration is periodically saved in order to allow for bump less restarts.

**Addressing**

Via the NOMAD tree view and by name

**Simulation mode**

Implemented via a separate scheduler in the NOMAD server. Several different simulation modes exist.

**Logging**

Done centrally by the NOMAD server in either text or XML files. Several logs exist: a user log and a debugging log.

**Debugging**

There is a debugging log. It is also possible to have a live view into the NOMAD server in order to inspect processes and variables. There is automatic crash reporting.

**Error propagation**

Through CORBA event notifications.

**Remote access**

In principle possible because of the client server architecture. In practice remote access is not being done for security reasons. Instrument scientists can connect via VNC. This is fast enough.

**Integration of sample environment and user hardware**

Known hardware is configured into the instrument. For unknown hardware there are communication primitives available from the scripting language to send commands along the known buses.

**Security**

The NOMAD server has a password system in order to differentiate three different users: user, instrument scientist and superuser.

**State Change Propagation**

Internally through links within the NOMAD server or through CORBA notifications.

**Parallel Processing**

Parallel processing happens through threads in the NOMAD server. A NOMAD supplied base class ensures safe thread execution within the NOMAD environment. A XML file is used to define what can be run in parallel. Otherwise there exist per object locks. Operations can be collected into a block. When this block is active then all participating objects are locked. Thus the block serves as a single synchronization point. For all hardware parameters there is a target, value division which avoids read/writes at the same time. Stopping, pausing etc. is implemented via state variables.

## *Management Issues*

| ***Activity*** | ***Time required*** |
|---|---|
| Teaching a user | 0.5 h |
| Teach an instrument scientist | 0.5 day |
| Teach device development | 1 week |
| Induce new IT staff | 1-2 months |
| Develop drivers | 0.5 day |

ILL runs 40 instruments with 6 people for software support and development. Besides the neutron instruments at ILL, NOMAD is also used on two beam lines at CSNSM, the Centre de Spectrometrie Nucleaire et de Spectrometrie de Masses, Universite d'Orsay, Paris.

# EPICS

## *Overview*

EPICS is a control system architecture which is very popular for use with accelerators. Increasingly it is being used for slow controls at instruments too. EPICS is a large collaboration including: APS, SLAC, SLS, DESY, BESSY, SNS, Fermilab, the australian synchrotron and more.

EPICS scales very well. Setups with several hundred IOCs (servers) are common. Many clients can connect to each IOC without any performance loss.

EPICS has something to offer for all three layers of data acquisition:

## Layer 1: Clients

- Thin clients are provided:
    - Drag&Drop display builder (medm, edm, ...)
    - Value vs time curves (StripTool)
    - Alarm handler
    - Archiver
- APIs for
    - C/C++
    - Java
    - Tcl/Tk
    - Python
    - IDL
    - Matlab
    - LabVIEW
    - many other languages
- Thick clients can be implemented with these APIs but are usually not recommended.

## Layer 2: A Network protocol: Channel Access

Provides read and write access as well as event subscription (callback whenever a value has changed significantly)

## Layer 3: Servers

EPICS servers are called IOC, for I/O controller. One central concept of EPICS and of course an EPICS-IOC is the EPICS database. An EPICS database consists of different types of records. A IOC manages a configurable set of records. Each records holds a set of process variables, PV. The server component of the IOC sets PV's in the record via the channel access protocol. A configurable per record scanner process or thread then makes the changes to the record actually happen.

## *System Requirements*

IOCs run on vxWorks and RTEMS real-time operating systems as well as on Linux (embedded Linux too, but maybe not CLinux), MacOS, other UNIX-like systems and Windows (at least XP and 7, but not CE). Real-time scheduling is used if available by the OS. IOCs are written in C/C++.

There are EPICS installations which do not use commercial software. On the other hand ~30% of all drivers still require vxWorks.

Any Processor architecture that is currently supported by one of these operating systems should work with EPICS, either out of the box or after minor configuration work. Both, little endian (Intel) and big endian (Motorola) processors are supported in 32 bit and 64 bit versions.

Clients run on the same types of operating systems. Many GUIs still use the Motif graphics library, but a move to QT is visible. Also Java clients become more and more popular. A client toolbox based on Eclipse (Control System Studio, CSS) is under active development.

Generally, support for Linux and other Unix-like operating systems seems better than for Windows.

The network protocol is based on IPv4 TCP and UDP broadcasts.

## *Hardware Access*

All hardware access happens via IOC's. Drivers have a defined interface to records: A function table with init, read/write, report, and other functions. This table is registered with a string name by which it is found by a record that wants to use it. The record provides a configuration string that is used by the driver to address the hardware. Drivers are written in C or C++.

Drivers read or write single values, arrays or strings. For complex instruments, a driver often has to handle many values (connected with many records) and requires knowledge of the internal of a device. This is usually implemented with instances of a driver that handle one device each. Therefore driver code needs to be written with re-entrancy and multi-threading in mind.

Typically a driver is divided into two layers: an lower layer that handles hardware access and an upper layer that deals with records.

There is a framework for drivers which provides several features that are commonly used by drivers, such as work threads, access sequencer, lock mechanism, etc. This framework also abstracts hardware in a way that makes record configuration more generic and uniform across different hardware implementations. Hardware is abstracted as either "registers" (of several data types) or "byte streams" (e.g. GPIB, Serial, telnet like tcp, ...) or a combination ob both. This makes it possible to use a common upper layer for a wide range of devices. This framework is more and more used by new drivers. But there is still a big set of old drivers that are not coded to the common driver framework.

For simpler devices which follow a command response protocol there exists a streaming infrastructure which allows to configure the formatting and parsing of the commands and responses. With this tool many drivers can be developed through configuration without writing C code.

EPICS does not have a device model. It models individual values. Each value has a name by which it is addressed and which must be unique on the network. EPICS values can have attributes which allow to define units, valid ranges and most importantly error conditions. All EPICS values have a time stamp.

Another thing to keep in mind is that the EPICS IOC has a relatively small output buffer for clients. In order to limit the data rate to clients the EPICS IOC can choose to overwrite values in the output buffer by newer ones. Thus a client may not get all intermediate values of a parameter. This is very OK for a control system where you are always interested in the latest and best parameter. But may lead to problems in a DAQ system.

There are also EPICS records which implement functionality such as physical values mapping like wavelength and a flexible scan record.

### *Neutron Data Acquisition*

EPICS does not have special facilities for neutron data acquisition except for a 1D histogramming record. EPICS only supports 1D arrays of a fairly static size. Multi dimensional arrays and actual array lengths must be handled by convention.

### *Implementation of Higher Level Functionality*

The recommended way to develop applications in EPICS is through EPICS real time database configuration. EPICS does not only provide database records for hardware but also for control flow, logic and calculations. Further high level records implement things such as scans, monochromators or other high level functionality. There exists a popular calculation record which essentially allows to call to a user defined C-function.

Database records can be connected via links. Links allow events from one record to propagate to other records, possibly in different IOC's. Linking thus allows to create a data flow style network of records.

For problems too complex to be solved with interconnected records, a finite state machine engine is provided that runs on the IOC. It is programmed in a C-like language. It reacts on changes of records (typically on the same IOC) and writes to other records.

Addition or deletion of records requires a restart of the IOC, as well as changing the connection of I/O records to hardware. However, re-configuration of existing record can be done during operation. This is the usual way to set up the data acquisition for a specific measurement. Since these configurations are values like any other parameters, they can be stored to files and restored later.

Since inter-connection of records via links works (almost) transparently over the network (except for timing issues), it is often possible to separate the control flow logic from the hardware access. This makes it possible to reprogram the logic without losing control over the hardware during reboot.

With all these features the EPICS real-time database can be thought of as a programming system in its own right, much like FPGA or PLC programming. It should be clear by now that the EPICS database is more like a set of distributed records and has nothing to do with a database in the traditional sense which supports tables, SQL and much more.

Client side logic is possible but frowned upon. For prototyping it is a quick way to program something in the preferred (script) programming language. However, this creates a dependency of the instrument on the client program. It must be assured that the client program runs all the time and runs only once. Thus, it is preferred to put the logic into the IOC.

Summarizing this section: the EPICS approach is to configure and join multiple components (EPICS database records) together rather then to write code. Writing code is an error prone activity, thus this approach has appeal.

## *User Interfaces*

On top of the EPICS database no standard system exists. Most instruments control their experiments through scripts written in one of the supported scripting languages.

There exist configurable tool kits for developing tailor made graphical user interfaces, most notably MDM and control system studio (CSS). These tool kits are designed to fulfill the needs of accelerator operators. Accelerator operators are usually well trained staff who essentially do the same operations all over at a fairly static machine. For the more dynamic environment at instruments, where hardware and procedures change rapidly, these tool kits are of limited use.

EPICS can be scripted in many scripting languages. For most scripting languages a channel access binding exists.

## *Control System Features*

**Configuration**

Via text files distributed across the IOC computers. Use of version control for these files is recommended. Configuration files for clients usually live on a shared network file system. Save options at the IOC cater for value storage to allow bump-less restarts.

**Addressing**

Each parameter *MUST* have a unique name. The naming must be defined by convention.

**Simulation mode**

Can not easily be implemented. A duplicate complete network of IOCs with simulation hardware has to be set up. And kept in sync with the real instrument.

**Logging**

A variety of configurable logging options exists. From logging to local files at various levels of detail to logging to a central logging server. There is an archiver tool which can log the values of a selected set of parameters over time.

**Debugging**

No special debugger exists. On the hardware level debug messages can be switched on. On the IOC real-time database level, semantic errors based on e.g. spelling errors can be found relatively easily with the help of built-in diagnostics. Logic errors are much more difficult to find, because everything seems to work fine for the computer, it is simply doing the wrong things in the opinion of the user. Since every value, including intermediate results, in the system is readable and can be monitored, functionality of the real-time database can be checked against ones expectations. Some problems can be diagnosed with the built in alarm system.

**Error propagation**

EPICS forwards alarm states together with values across links. But in the end the client program has to check if an alarm condition is present on a parameter.

**Remote access**

If there is access to the network on which the EPICS real time database is implemented then everything can be done.

**Integration of sample environment and user hardware**

This is easiest if a EPICS driver exists. Or by feeding into well known hardware I/O interfaces.

**Security**

Read/write protection on a per user/host level can be configured. Though in practice security is often controlled through network access rules.

**State Change Propagation**

Either via record links or via notification through the CA protocol.

**Unit Testing**

Not many unit tests exist.

**Parallel Processing**

EPICS is highly parallel. Data integrity is ensured at the IOC level.

## Management Issues

| *Activity* | *Time required* |
|---|---|
| Teaching a user | 0.5 h |
| Teach an instrument scientist | 1 day basics |
| Teach device development | 6-12 months |
| Induce new IT staff | 3-6 months |
| Develop drivers | days to months |

Becoming a real EPICS expert can take years. EPICS is a big and old collaboration. Thus a lot of solutions exits for many problems. The real strength of EPICS is its hardware support: drivers exist for many hardware components.

SLS has 8 staff for 20 SLS beam lines, supported by a separate system administrators group and 2-3 people in another group writing drivers. This level of staffing is inadequate and causes frustrations with scientists.

## EPICS-4

A new version of EPICS, EPICS-4, is under development. EPICS-4 is designed to be interoperable with the current version of EPICS, EPICS-3. EPICS-4 brings the following new features to EPICS:

- Complex data types. Whereas EPICS-3 had no device model to group parameters together, this will be possible with EPICS-4.
- Some limitations regarding data types are lifted.
- Much improved support for arrays. Where EPICS-3 only supported static arrays, EPICS-4 will support dynamically sized arrays. EPICS-4 will also allow to send differences only when only a small part of an arrays changes.
- In order to support the new feature there is a new network protocol, PVAS. Channel access will continue to be included.
- Data rate limitations will be per client rather then per IOC.
- EPICS-4 aspires to add a middle layer on top of EPICS-3, a service oriented architecture (SOA). This will allow a more RPC style interaction with the control system. The programming model is optimized for efficient distributed synchronous or asynchronous processes communicating by essential a shared-memory system based on data introspection.

The status of EPICS-4 as of 10/2012 is as such:

- The SOA and structured data components are in beta and are being tested. This is implemented on top of existing EPICS-3 IOC's.
- Full replacement for EPICS-3 style IOC services will take some more years to come along.

Thus EPICS-4 is not mature yet (and will not be for some time) but is an interesting development to be watched.

## Data Acquisition at SNS

At the time of this report, 2012, DAQ at SNS is in transition. Both the neutron data acquisition software and the system used for slow control in the initial implementation of SNS instruments is being replaced. In this section we will focus mainly on what has been learned concerning high data rate event mode data acquisition at SNS.

Enough of the old control system will be described to provide a background for the decision to redo the system. And for the lessons learned from this system.

The new slow control system will be based on EPICS. Currently the ideas are being tested on a HIFAR imaging beam line. This system is not ready yet. Thus the current state as of 7/2012 will be described.

### *Neutron Data Acquisition*

SNS uses He3 tube detectors and scintillator detectors. In order to interface these detectors a modular set of electronics components is used. There are three classes of boards:

- Data acquisition boards for interfacing to different detector types and external high speed I/O
- Calculation boards for calculating positions and pixel ID's
- Storage boards for doing event processing and communication

The communication between these boards happens via LVDS. For communication with the data acquisition preprocessor computer and the timing system an optical link is used.

All these boards form a network which essentially performs the following tasks:

- For each detected neutron event generate a event packet consisting of a four byte pixel-ID and a four byte time stamp relative to the last pulse. The pixel-ID is unique within the instrument.
- Generate additional events which indicate a pulse start
- Distribute timing information and DAQ commands to all participating boards as required.

There are strong forces which make this event handling scheme mandatory. In order to transfer the maximum amount of neutron events across communication links it is advisable to minimize the amount of data to transfer per neutron event. The combination of pixel-ID and a relative time stamp plus pulse events is this minimum. A more complete event with an absolute time stamp and more detailed pixel coordinates would be advantageous for downstream processing but is not feasible performance wise.

The handover to the data acquisition software happens in the preprocessor computers. These read the data from the electronics through the optical link and perform further operations on the raw event data. There may be more then one preprocessor computer in order to cope with high data rates.

## *Old Style Neutron Data Handling*

This scheme uses the following steps:

1. The preprocessor computer stores the events data in a raw binary file after preprocessing.
2. The raw file is transferred to another machine for further analysis
3. The next step is the generation of an event NeXus file from the raw event files and meta data.
4. Then the event NeXus file is analyzed.

For a two hour experiment this process could take up to two hours until the user was able to look at her data. Users are dissatisfied with this waiting period.

## *Second Generation Neutron Data Handling*

In order to give users faster feedback on their data a new system was developed. This system, ADARA, consists of the following components:

- A Streaming Management Service (SMS)
- A Streaming Translation Service
- A Streaming Reduction Service
- A histogram service

The heart of the new system is the Streaming Management Service (SMS). The SMS:

- Collects neutron event data from the preprocessor computers
- Collects auxiliary inputs from slow controls and the DAQ system
- Packages all that information into a combined event data stream
- Forwards the combined event data stream via TCP/IP to connected clients

One notable client to the SMS is the Streaming Translation Service. This service receives the data from the SMS and writes it to an event NeXus file on a parallel file system.

The Streaming Reduction Service is basically a copy of Mantid which reads the combined event stream from the SMS and directly corrects and reduces the data. Mantid is a component based data analysis framework developed jointly between ISIS and SNS. Mantid has been heavily parallelized in order to enable it to process more then 12 Million events/second. And there is still room for further improvements.

For providing immediate online feedback on data collection as histograms a differently configured copy of Mantid is used which reads the event stream directly from the SMS.

## *Event Mode Data Reduction*

Before reducing neutron data to physical quantities some correction need to be applied. Usually this is a combination of:

- Scaling to correct for detector efficiency differences and different monitor counts

- Subtraction of background or empty can runs

Both are surprisingly easy solved in event mode:

- For scaling, a weight is applied to each neutron event.
- For subtraction the background event stream is added to the real data with negative neutron events.

## SNS Data Rates

In order to get a better estimate of the data rates to be expected at ESS it is good to look on some numbers from the SNS

- SNS is now operating at around two thirds of its design power
- Most instrument actually produce neutron events in the order of 20-30 K neutrons/second. This is not surprising: with a next generation neutron source smaller samples are studied or more resolution is aimed for. Both experimental conditions reduce the amount of scattering.
- Once the background is down it requires less disk space to store event mode data rather then histogram data. This is especially true for very large detector arrays and weak scattering. Then a histogram will contain many zeros and thus be a sparse matrix.
- Only few SNS instruments generate high data rates. NOMAD is the strongest producer with 640 MB/second.
- The total data rate for all SNS instruments averages out at 4GB/second.
- The current data production at SNS is 150TB/year. This is expected to rise to 300TB/year.

## PyDAS and old slow control

The current SNS control system is based on Windows-XP computers, Labview and python. There are separate computers for each controller which operate the actual hardware via Labview. A central data acquisition computer maps all those device computers in shared memory. The connection to the device computers happens via national Instruments data sockets. On the central DAQ computer runs the PyDAS software. PyDAS is the actual control program which controls motors, other devices and the neutron DAQ. It runs scans and batch files and displays online data. PyDas is a python application.

There are various problems with this setup:

- There are 12-15 Windows computer per beam line. Maintenance and system updates are cost intensive.
- The solution promised to use commodity hardware. In practice it turned out that some hardware cards only work with certain mother boards. Thus special computers needed to be acquired anyway.
- The connection to the device computers via shared memory implements a strong coupling between server and clients. This caused maintenance problems.
- NI data sockets perform well most of the time. But sometimes they do not and it is not clear what is causing the problem.

## *New EPICS Slow Control*

Following an organizational reshuffling and and a review of the existing control system, SNS decided to implement a new control system. This is currently being prototyped at a neutron imaging beam line at HIFAR. The new setup will use EPICS in order to communicate with devices. EPICS was decided upon because the SNS accelerator division is already using EPICS and thus has the necessary expertise to support the move. On top of EPICS there will be a SNS developed scan server which actually performs the measurements. The SNS scan server maintains a queue of scans to perform. Scans are broken down to primitives by top level code.

On top of the SNS scan server there will be an instrument specific graphical user interface developed in Control System Studio (CSS), an EPICS UI generation tool.

As already said, the new system is currently being prototyped. Depending on the success of the prototype the final setup may still change.

## Data Acquisition at ISIS

The DAQ system at ISIS consists of the following components:

- A central instrument control computer running Windows 7, 64 bit
- Most hardware is accessed via Labview
- Neutron data acquisition is different: there is a C++ program which interfaces with the detector electronics and is responsible for writing data files.
- There is a central control application, SECI, which is used to manage the Labview components and the neutron data acquisition system.

ISIS performs experiments both in histogramming and event mode.

ISIS is currently undergoing a review of their data acquisition system. EPICS is currently being actively evaluated as a new basis for the DAQ system. This is currently in a prototype stage.

### *ISIS System Requirements*

The ISIS DAQ system requires:

- Windows 7 64 bit
- Labview
- C++ for the neutron acquisition
- C# (C-Sharp) for the user interface

Interestingly, ISIS uses Windows 7 not on a plain computer but runs the DAQ server in a virtual machine. There is one server class computer for running virtual machines per instrument. The advantage is that virtual machine images can be easily backuped or copied to other physical computers. Also, in the case of an upgrade or modification, new software can be installed in a new virtual machine. Testing the new software or returning to a known good state then just involves starting the appropriate virtual machine.

### *ISIS Hardware Access*

At ISIS the preferred way to physically access hardware is via TCP/IP. Controllers which do not natively speak TCP/IP but other protocols such as RS-232, USB, GPIB etc are accessed via TCP/IP bridges. Such remote I/O ports are mapped via appropriate Windows drivers to local ports in such a way that the server software can assume to work with hardware connected to the local machine. On the software side Labview drivers and VI's (Virtual Instruments) are used to communicate with the hardware. ISIS had to develop many Labview drivers themselves, even when vendor supplied drivers existed. There are two reasons for this surprising fact:

- One reason is the low code quality in the vendor supplied drivers.
- The other reason is the different use case for a driver in an instrument control system. In a normal Windows environment it is perfectly acceptable to put up a dialog box when something needs to be configured or an error needs to be acknowledged. In an instrument control system the desired behavior is to get required information from external storage and to log errors somewhere and to keep going if at all possible.

As a side note it is worth to note down ISIS experiences with using Labview. Labview is very powerful. Recent versions of Labview contain enough features to write well structured Labview programs: component structures, object oriented programming facilities, message queues and controlled variables for multi threaded programs, event handling and much more. Even fine grained control of user interfaces is possible. On the other hand Labview is a very forgiving environment. Labview makes it easy to get away with bad code which just barely works. Concluding this paragraph, part of the bad reputation of Labview in the community is due to bad Labview programmers. Labview is just a language which can do great things when used properly. But reservations because of the commercial lock in are still valid.

## *Neutron Data Acquisition*

In order to understand ISIS neutron data acquisition it is necessary to understand the electronics. The current system is called DAE-2 with an upgrade to a more powerful DAE-3 system being work in progress. ISIS uses a modular electronics system based on VME cards. There are basically 3 types of cards:

- A communication card
- An environment card
- Detector cards

The communication card is responsible for the communication with the electronics. It implements the configuration of the electronics and DAQ and downloading of data. Currently this works via USB and is limited to 5Mb/second for DAE2. The newer electronics, DAE3, has a 1GB ethernet link and work is in progress to upgrade the DAE3 link to 10GB ethernet.

The environment card handles timing and veto signals. It receives the pulse signal from the accelerator and maintains a frame (pulse) count. It also receives veto signals from wherever this is necessary. The environment card propagates the pulse and veto signals across the VME bus. At this point it is necessary to explain how time of flight is handled at ISIS. At each neutron pulse a frame start event is sent with an absolute time stamp. Neutron event time stamps are relative to this frame start event. Thus the absolute time of flight value of a neutron events is calculated from:

```
Absolute neutron event time = Absolute frame start time stamp  +
relative time stamp
```

The environment card supports various modes of dealing with pulses: it is possible to have multi period frames (pulses left out), secondary pulse input (from a chopper) etc.

The real work of neutron event processing is done at the detector cards. As their input detector cards connect to detector input cards (DIC). These cards are detector specific and provide the position information for the neutron events. The actual time stamping is done by the detector cards. The default mode of operation of the detector cards is histogramming. To this purpose the detector card has 128MB worth of memory per card. By using as many detector cards as necessary the electronics can scale to big detector systems. If the need arises detector cards can be chained across different VME crates via VME extenders. The detector cards use 16 bit for time stamps and can handle time ranges from .5 to 162.5 nanoseconds. The maximum neutron rate per detector module in histogramming mode is 16MHZ. The histogramming done by the detector cards is configurable.

In event mode, the system works slightly different. Then raw neutron events are stored together with a frame header in the histogram memory area. Event data is continuously read out via the data link.

On the software side the DAE-2 electronics is handled by a C++ program. This program

- Configures the detector system
- Starts and stops data acquisition
- Reads the neutron data from the electronics
- Writes NeXus data files

Meta data and log data to be stored in the NeXus file is retrieved from the instrument manager SECI via DCOM. Instruments either write TOF-RAW or muon NeXus files. The full NeXus instrument structure is not implemented. In event mode the application writes NeXus event mode data files, compatible with those of the SNS. Thus data can be reduced with the Mantid package.

## *Higher Level Functionality and User Interfaces*

On top of Labview ISIS uses the instrument manager SECI. SECI is a graphical C# application. The SECI user interface provides two main areas:

- A Dashboard area. This is an area where important instrument information is displayed. There is a static dashboard part and a user configurable section where users can choose to display labview variables.
- A second area where SECI acts more like a window manager and displays the hierarchy of labview VI's. SECI can configure, start and stop the Labview VI's

Other features of SECI include:

- Users can select Labview variables for logging during the experiment. Logs are stored in a SQLite data base
- SECI also contains an error log view
- Creates a read only dashboard WWW-page
- Control of data acquisition
- Limited facilities exist to display online data while the data is being collected at the instrument

Scripting is possible from openGenie and python. Scripting is implemented by accessing VI's in SECI via DCOM. A small interface layer exists which simplifies DCOM access for scripting languages.

## *ISIS Control System Features*

### Configuration

XML configuration files which configure SECI for different instruments. There is inheritance in configurations in order to allow for different configurations of a given instrument.

**Addressing**

Addressing happens via Labview labels.

**Simulation mode**

Not really provided for. When scripting in openGenie the byte code compiler provides some syntax checking

**Logging**

Done by SECI at various levels

**Debugging**

Labview has powerful debugging tools

**Error propagation**

Logged in SECI

**Remote access**

Only through remote desktop

**Integration of sample environment and user hardware**

Via Labview device drivers. Most sample environment comes with Labview support.

**Security**

No restriction in the DAQ system. Windows access rules and common sense.

**State Change Propagation**

As SECI is the only program used not necessary. Scripts need to poll.

**Parallel Processing**

Labview can multithread. The neutron data acquisition runs in a separate process.

## *Management Issues*

| *Activity* | *Time required* |
|---|---|
| Teaching a user | 10 min |
| Teach an instrument scientist | 0.5 day |
| Teach device development | 1 week |
| Induce new IT staff | 3-6 months |
| Develop drivers | From a few hours to weeks |

ISIS currently operates 33 instruments with 7 staff of which 5 program. Of these 3 are Labview programmers. External staff is employed occasionally mostly for writing Labview drivers.

This system is used at ISIS only.

## DAQ at Diamond and the GDA

Diamond uses a Java control system with the name GDA on top of EPICS for data acquisition. EPICS has been described elsewhere in this document. Thus this section will mainly focus onto the GDA.

The first interesting part however, is the distribution of labour between GDA and EPICS. Diamond uses EPICS as a means of hardware device control. All upper level functionality is implemented in the GDA. There is a small overlap in both directions as some more advanced EPICS tools are used as well as that the GDA implements some less important drivers directly.

The GDA itself consists of a Java server process and an Eclipse-RCP client application.

## GDA System Requirements

The GDA is written in Java and thus largely operating system independent. But development focuses on the Linux and Windows platforms.

## The GDA Server

The GDA server is a plain Java application, not an Eclipse-RCP application. It does run the experiments, is responsible for data file writing and such. It includes a jython scripting interpreter. The GDA server is configured via a XML configuration file using the Spring framework. Spring is a dependency injection tool. Spring reads an XML file. The XML file describes a framework of java objects which are then put together by Spring to form an application. Usually there is only one GDA server per instrument. If necessary, the GDA server can call out to additional external servers via CORBA or RMI. Some features of the GDA server:

- Role based access control
- Baton system to control which client runs the instrument
- Editable queue of scans to perform
- Powerful scan system

## GDA Clients and Communication with the GDA Server

The GDA client is an Eclipse-RCP application composed of plugins. The GDA client contains a number of tabs which allow to view the status of the instrument, edit scans, interact with the scripting interpreter and more. Thus the GDA client can be seen as a pure parameter and experiment editor. Some instruments have instrument specific tabs. The GDA client is configured via Spring XML files and property files. The GDA client also contains a NeXus file viewer.

Communication with the GDA server happens via CORBA or RMI. The GDA server only passes state changes up to the GDA client, no values. Thus the GDA server informs the client when for example a motor starts or stops but not about the position of the motor. For values, the client has to use EPICS channel access. Thus, the GDA uses three network protocols: Java RMI, CORBA and EPICS channel access.

## *GDA Features*

**Configuration**

XML configuration files for the Spring framework

**Addressing**

Names are defined in XML files. There is a finder service which allows to locate devices.

**Simulation mode**

Simulation mode in a separate GDA server

**Logging**

Logging configurable at object level in the GDA server

**Debugging**

See logging. Many instruments have special jython scripts which test the health of the instrument.

**Error propagation**

EPICS errors and other errors are passed through the system as exceptions.

**Remote access**

Via NX client

**Integration of sample environment and user hardware**

Via EPICS drivers. There is also the possibility to write drivers in Java or access devices via jython scripts.

**Security**

Role based access control, baton system and cooperation of users

**State Change Propagation**

Through CORBA and EPICS CA

**Testing**

High coverage with unit tests

**Parallel Processing**

By using EPICS for hardware access, there is already some parellisation. The GDA server can call out to external services when necessary. Control is maintained via state variables.

## *Management Issues*

| ***Activity*** | ***Time required*** |
|---|---|
| Teaching a user | 30 min |
| Teach an instrument scientist | No data |
| Teach device development | see EPICS |
| Induce new IT staff | 3-6 months |
| Develop drivers | From a few hours to weeks |

Diamond currently operates 31 beam lines with 12 EPICS control engineers and 12 staff looking after GDA. In addition there are 8 people doing data analysis software. Building the GDA client application is so complex that a special build, test and release engineer is required.

GDA is designed to be developed collaboratively. However, as of now it is only used at Diamond.

## IROHA

IROHA is the summarizing name of the DAQ middle ware at KEK/JPARC in Japan. IROHA is a distributed control system based on a component model. The DAQ component model was based on a standard robotics component model called Robotics Technology middleware. This standard is discussed and defined by the Object Management Group (OMG). A RT-component offers typed input and output data streams and service interfaces for controlling and interacting with the component. A RT-component also contains a state machine. RT-components can be interconnected with each other in data flow networks. For the purpose of DAQ, JPARC/KEK has extended this component model slightly.

Initially CORBA was used for inter component communication. This has been replaced by XML messages across HTTP. This is a RPC style communication.

### *System Requirements*

IROHA runs on Linux.

### *Hardware Access*

Hardware is accessed through RT-middleware components.

### *Neutron Data Acquisition*

At the electronics side neutron data acquisition is handled by NEUNET VME bus cards. The NEUNET cards handle both histogram and event mode data collection. The NEUNET cards run a special network protocol SiTCP on top of standard ethernet hardware. These NEUNET cards forward the collected neutron events to a DAQ unit. On this DAQ unit four other components process the neutron event data:

- A gatherer component reads the neutron events and forwards them to the dispatcher
- The dispatcher forwards the neutron event stream to the logger and the monitor
- The logger saves the event stream to disk into NeXus files
- The monitor component histograms the event data for online display

64 bits of data are transferred by neutron event: 24 bits TOF-time, 8 bits PSD number, 12 bits right and left pulse height. In addition there are pulse time events. Reconstruction of absolute time happens by keeping track of T0 frames which are sent whenever a neutron pulse is generated.

Data analysis of event data happens through manyo-lib programs.

### *Implementation of Higher Level Functionality*

Higher level functionality can be implemented either as RT-middleware components in C++ or through scripting in python.

## User Interfaces

Instruments have instrument specific graphical user interfaces. In terms of common user interfaces there is a launcher, a sequencer for data analysis, an experiment scheduler for batch processing and scripting, a status monitor and instrument status windows.

The experiment scheduler provides for scripting in python.

## Control System Features

**Configuration**

Via XML files. JPARC/KEK is moving towards storing the configuration in a XML database.

**Addressing**

You need to know the URL's of RT-middleware components

**Simulation mode**

None

**Logging**

To XML files at component level

**Debugging**

Some components have debug modes. There is also a test bed to test the system offline

**Error propagation**

By polling RT-middleware components from the DAQ operator console

**Remote access**

No remote access foreseen as of now. Remote access via VPN is in consideration.

**Integration of sample environment and user hardware**

For sample environment devices RT-middleware components are provided. Special support for supporting user hardware does not exist yet.

**Security**

Security is achieved through network security: each instrument lives in its own VLAN, separated by firewalls.

**State Change Propagation**

By polling RT-middleware components from the DAQ operator.

**Parallel Processing**

Components run independently. For neutron data acquisition many pipelines may run in parallel.

## *Management Issues*

| ***Activity*** | ***Time required*** |
|---|---|
| Teaching a user | 2-3 h |
| Teach an instrument scientist | 1 months |
| Teach device development | 12 months |
| Induce new IT staff | 12 months |
| Develop drivers | 3 months |

KEK/JPARC does not have a data acquisition computing group in its own right. DAQ software and hardware are developed between the electronics group and scientists.

## General Experiences

During the survey for this control system comparison questions for more general experiences and what people would do different given a second chance were asked. This section contains a selection of answers.

### *Technical Experiences*

- In general the control systems surveyed were themselves the result of an evolution. During this evolution many initial design weaknesses were fixed.
- Make sure to have a command line interface
- Several people like to move away from CORBA. Reasons include:
  - Inefficient
  - Difficult to maintain
  - Complicated
- The client server architecture has proven its value
- Use a system which allows to easily modify GUI's
- Hardware standardization is a good thing
- It is a good idea to make parts of the system replaceable. This allows to fix wrong technical choices or adapt to newer technologies. This requires to abstract a number of certain services in the system to allow for the possibility of different implementations.
- Python is the preferred scripting language.
- Network protocols used in DAQ need to cover two requirements:
  - Write/Read RPC style interactions
  - Notification messages about state changes

### *Organizational Experiences*

- The organization needs to avoid blame games: the user blames the software, the software engineer the electronics etc. This does not solve problems.
- Take care of data formats and management and other standards right from the start. It is difficult to resolve a mess once it has occurred.
- Having a common sequencer on top of a distributed system helps. Otherwise there is the need to maintain user scripts in many different languages/formats at the instruments. A shared sequencer can also take care to write coherent data files with all necessary information. And it also provides a coherent user experience across instruments.
- It is important to have a good and cooperative work environment
- A uniform system is important, almost independent of the actual choice.
- Integration and interfaces must be enforced
- Being part of a collaboration can also have its downsides: the need to will arise to maintain code which has been written at different places with different code styles. Or which may be hard to understand for other reasons.

## Common Patterns

During this survey some patterns shared among control system could be identified. These are listed in this section. Some of them seem trivial. But it is nevertheless worth to make a note of them.

**State variables for controlling asynchronous operations**

When doing operations asynchronously, the client side is interested to figure out if the requested change is still in progress or has already finished. To this purpose, state variables are used which go through a sequence of idle, running, idle when processing. A separate variable holds the success or error of the operation.

**Target/value separation for hardware variables**

For hardware variables where a requested change must not necessarily be obeyed there are often a pair of variables: a target and a value. The value is the actual value as read back from the device, the target is the one requested. This is also helps to separate thread access: the target is only set by the client, the value is set only by the device server.

**Layering of hardware access**

In hardware access there is often a layering: a bus layer, a driver framework and then the actual driver. The aim is that the actual driver only needs to concern itself with formatting the necessary commands for the device and parsing replies from the device and the devices very own logic (or unlogic) and not so much with infrastructure.

**Abstraction of system services**

Some systems provide an abstraction on top of system services such as threads, network protocols, memory etc. The rationale is to change only the library and not valuable domain code when the underlying technology changes.

**Higher level functionality implemented on top of intermediate language**

Some systems choose to decompose higher level constructs such as scans into a set of primitive commands to be executed as a block by an executor. The advantage is the reuse of the executor. The other advantage is that different execution modes, such as simulation modes can easily be implemented by changing the executor.

## Data Acquisition System Feature Comparison

A comparison of the features in the various DAQ system can be seen in the table below:

| **Feature** | **Tango** | **Sardana** | **NOMAD** | **EPICS** | **GDA** | **SNS** | **ISIS** | **KEK/JPARC** |
|---|---|---|---|---|---|---|---|---|
| Architecture | DC | SC on DC | SC | DC | SC on DC | SC | SC | DC |
| Preferred OS | Linux | Linux | Linux | Linux | None | Windows | Windows | Linux |
| Required SW | CORBA | CORBA | CORBA | None | Java | Labview | Labview | None |
| HW access | DS | DS, direct | direct | DS | DS, direct | Labview | Labview | RT-componen |
| NW-Protocoll | CORBA | CORBA | CORBA | CA | CA, CORBA | Labview-DS | DCOM | HTTP/XML |
| Scripting | Many | Python | MAD | Many | Jython | Python | Python, OG | Python |
| Standard GUI | Many | taurusgui | NOMAD | Many | GDA |  | SECI | Many |
| Neutron DAQ | No | No | HM | No | No | Stream | HM, stream | HM, stream |
| Configuration | Tango-DB | Tango-DB | XML | Distributed | XML |  | XML | XML |
| Addressing | Path | Path | Names | CA-PV | CA-PV, name | Names | Names | URL |
| Simulation | None | Yes | Yes | None | Yes | None | None | None |
| Logging | Various | Various | XML | Various | Various | Local | Local | XML |
| Propagation | 0MQ | 0MQ | CORBA not | CA | CA,CORBA | Labview | Labview | Polling |
| Remote | VNC | VNC | VNC | VNC | VNC | VNC | VNC | None |
| Parallel | Yes | Yes | Threads | Yes | Yes | Limited | Threads | Yes |
| Support | 8 F | 2 F | ILL | Many F | Diamond | 1 F | 1F | 1F |
| User intro | 2-3 h | 2-3h | .5 h | 0.5 h | 30 min | N/A | 30 min | 2-3 h |
| Instrument SC | 1 months | 1 months | .5 day | 1 day | N/A | N/A | day | 1 months |
| Device dev | 3 days | 3 days | 1 week | 6-12 month | 6-12 months | N/A | 1 week | 12 months |
| IT staff | 6 months | 6 months | 1-2 month | 3-6 month | 3-6 months | N/A | 3-6 month | 12 months |
| Driver dev | Days – weeks | Days – weeks | days | days to month | h to weeks | N/A | hours to weeks | 3 months |

Legend: DC: Distributed Control System, SC: single kernel control system, DS: device server, CA: channel access, HM: histogramming, 0MQ: ZeroMQ, F: facilities,

## Actual Manpower Used for DAQ

This little table compares the number of DAQ staff with the number of instruments supported.

| *Facility* | *DAQ-System* | *DAQ-staff* | *No Instruments* | *Staff/Instrument* |
|---|---|---|---|---|
| ESRF | TANGO | 17 | 42 | 0.4 |
| ILL | NOMAD | 6 | 40 | 0.15 |
| SLS | EPICS | 10 | 20 | 0.5 |
| ISIS | ISIS | 7 | 33 | 0.21 |
| DIAMOND | EPICS/GDA | 24 | 31 | 0.77 |

Please take these numbers with care: in some cases they represent what the respective organizations are willing to provide and not what really is needed.

## Acknowledgments

I am deeply indebted to all those people to whom I have talked at the various facilities for advice and information which became the input for this comparison. Most notably (in no particular order):

- Andy Götz, ESRF
- Paolo Mutti, ILL
- Steven Hartmann, SNS
- Freddie Akeroyd, ISIS
- Takeshi Nakatani, MLF
- Paul Gibbons, DIAMOND
- Dirk Zimoch, SLS

## Glossary of Computing Terms

**IPC**

Inter process communication. Consider two or more cooperating process which have the need to exchange data with each other. This data exchange happens via IPC mechanisms.

**RPC**

Remote procedure calls. This is an IPC mechanism where functions in processes on different computers are called through the network.

**CORBA**

COmmon Request Broker Architecture. This is another IPC mechanism for data exchange between programs running on different computers. CORBA uses an object oriented approach: programs send messages to remote objects. CORBA provides services which allows to locate remote objects and for transporting CORBA calls. There are also code generators which generate skeleton server and client code from a description of the remote objects in IDL, Interface Definition Language.

**ZeroMQ**

This is another IPC mechanism where programs send each other messages which are stored in message queues. Participating programs then can inspect message queues for messages at their leisure and process them. Very flexible messaging schemes can be implemented with such queuing systems of which ZeroMQ is one implementation of.

**DCOM**

DCOM is a distributed component model developed by Microsoft. It is another IPC mechanism used mainly on Windows.

**RMI**

Remote Method Invocation. This is another IPC mechanism which is special to the Java programming language.

**RTOS**

Real time operating system. Normal operating systems such as Windows or Linux cannot guarantee that certain requests, especially HW requests, are processed within well defined time limits. Real time operating systems are optimized to do right that: guarantee processing within well defined time limits.

**LVDS**

Low Voltage Differential Signaling is a standard for high speed data transfer between electronics components.

**vxWorks**

vxWorks is a commercial real time operation system developed by the company Windrivers.

**VME**

VME is a popular bus system used to build custom computer systems for data acquisition. There exists crates (boxes) which have a back plane with the VME bus system. Computer and I/O and other cards can be plugged into this back plane to form a specialized computer system. The VME bus then takes care of data exchange between the different cards.

**Distributed Control System (DCS)**

A data acquisition system where the tasks of accessing hardware, calculation and other data acquisition services are distributed among different processes running on different computers. DCS excel when the number of hardware devices which need to be controlled becomes very large or when the system has to extend over large distances.

**XML**

XML is a structured ASCII file format. It is easy to process by computers and reasonably understandable by knowledgeable humans.

**Remote Desktop, VNC, NX**

Remote desktop systems are system which allow to control distant computers.

The whole desktop of the remote computer is transferred across a network link. VNC and NX are two popular implementations of remote desktop technology.

**Motif**

A toolkit for building graphical user interfaces. Motif is fairly old and is tied to the X-windows system normally installed on unix computers.

**QT**

QT is another toolkit for building graphical user interfaces. QT is a cross platform toolkit, i.e. code developed for one platform can be compiled for Windows, Macintosh and Linux systems.

**SWT**

SWT is a Java user interface development toolkit.

**Eclipse-RCP**

Means Eclipse Rich Client Platform. Eclipse is a development environment for programming in Java, C++ and many other programming languages. Eclipse itself uses a programming model where an application is built from a bundle of plugins. Each plugin contributes certain services or implements some functionality. The idea is that plugins can be reused between different applications. The infrastructure for making the plugin mechanism work is known as Eclipse-RCP and can be used to implement any type of application. Eclipse-RCP is bound to the Java programming language.