System Control Works

Concepts, Goals and Architecture

V0.7

Andrew Manison
February 13, 2026

1 INTRODUCTION 6

2 OVERVIEW 6

3 CORBA 8

3.1 CORBA SERVICES 8
3.1.1 NAMING SERVICE 9
3.1.2 TRADING SERVICE 9
3.1.3 EVENT SERVICE 9
3.1.4 NOTIFICATION SERVICE 10
3.1.5 TIME SERVICE 10
3.1.6 PROPERTY SERVICE 10

4 SYSTEM CONTROL WORKS COMPONENTS 11

4.1 VIRTUAL FUNCTION OBJECTS 11
4.1.1 NODE CONTROLLER VIRTUAL FUNCTION 11
4.1.2 SCRIPT ENGINE VIRTUAL FUNCTION 11
4.1.3 STATISTICS VIRTUAL FUNCTION 12
4.2 VIRTUAL INSTRUMENT OBJECTS 12
4.3 PHYSICAL INSTRUMENT OBJECTS 12
4.4 COMMUNICATIONS ADAPTER OBJECTS 13
4.5 USER INTERFACE MODULES 13
4.6 AUTHENTICATION SERVER 14
4.7 SYSTEM LOGGER 14
4.8 CONFIGURATION MANAGER 14
4.9 REVISION MANAGEMENT REPOSITORY 14
4.10 SYSTEM CONFIGURATION TOOL 15
4.10.1 LIST OF OBJECTS 15
4.10.2 STATIC INTERCONNECTIONS 15
4.10.3 OBJECT DEFAULTS 15

5 IMPLEMENTING COMPONENTS 15

5.1 THE OBJECT FACTORY CLASS 15
5.2 THE CORBA OBJECT SERVANT CLASS 16
5.3 THE OBJECT MANAGEMENT CLASS 16

6 MODULE INTERCONNECTIONS 16

6.1 STATIC INTERCONNECTIONS 16
6.2 DYNAMIC INTERCONNECTIONS 16
6.3 EXAMPLE OF INTERCONNECTIONS 16

7 SELF-REFERENCING MODULES 17

8 POINT OF CONTROL ARBITRATION 18

8.1 ACCESS CONTROL 18
8.2 SECURE CONNECTIONS 18

9 FAULT LOGGING AND RECOVERY 18

10 SYSTEM INITIALIZATION 19

10.1 SYSTEM STARTUP 19
10.1.1 SCW UNIVERSAL MANAGER 19
10.1.2 CONFIGURATION MANAGER 19
10.2 SYSTEM SHUTDOWN 19
10.3 REVISION MANAGEMENT 20
10.4 VERSION SWITCHING 20
10.5 FILESYSTEM HIERARCHY 20

11 SCW CLASS HIERARCHY 20

11.1 SCW BASE CLASS 20
11.2 SCW COMPONENT CLASS 20
11.3 INSTRUMENT PERSONALITY CLASSES 21

12 VIRTUAL INSTRUMENT CLASS 21

13 PHYSICAL INSTRUMENT CLASS 21

14 COMMUNICATIONS ADAPTER 21

14.1 BASE COMMUNICATIONS ADAPTER CLASS 22
14.1.1 MODULE FUNCTION 22
14.1.2 PROPERTIES IN SYSTEM CONFIGURATION 22
14.1.3 EMBEDDED OBJECTS 22
14.1.3.1 Virtual Circuit Class 22

14.1.4 METHODOLOGY 23
14.2 GPIB COMMUNICATIONS ADAPTER CLASS 24
14.2.1 MODULE FUNCTION 24
14.2.2 PROPERTIES IN SYSTEM CONFIGURATION 24
14.3 SERIAL COMMUNICATIONS ADAPTER CLASS 24
14.3.1 MODULE FUNCTION 24
14.3.2 PROPERTIES IN SYSTEM CONFIGURATION 24
14.4 TCP/IP COMMUNICATIONS ADAPTER CLASS 24
14.4.1 MODULE FUNCTION 24
14.4.2 PROPERTIES IN SYSTEM CONFIGURATION 25

15 OPERATIONAL SCENARIOS 25

15.1 SYSTEM STARTUP 25
15.1.1 MAIN SERVER 25
15.1.2 SLAVE SERVER 27
15.2 COMMUNICATIONS ADAPTER STARTUP 27
15.2.1 COMM ADAPTER OBJECT FACTORY 27
15.2.2 COMMUNICATIONS ADAPTER STARTUP 28
15.3 PHYSICAL INSTRUMENT STARTUP 28
15.3.1 PHYSICAL INSTRUMENT OBJECT FACTORY 28
15.3.2 PHYSICAL INSTRUMENT OBJECT 29
15.4 VIRTUAL INSTRUMENT STARTUP 30
15.4.1 VIRTUAL INSTRUMENT OBJECT FACTORY 30
15.4.2 VIRTUAL INSTRUMENT OBJECT 30

16 SCHEDULER VIRTUAL FUNCTION 31

16.1 SCHEDULER OBJECTS 32
16.1.1 TASK OBJECTS 32
16.1.2 RESOURCE OBJECTS 32
16.2 SCHEDULE CREATION 32
16.3 SCHEDULE PLAYBACK 33
16.3.1 MAIN SCHEDULER THREAD 33
16.3.2 JOB THREAD OBJECT 33

17 NODE CONTROLLER VIRTUAL FUNCTION 34

18 DESIGN PHILOSOPHY 34

18.1 PARADIGMS 34

19 EXAMPLE IMPLEMENTATION 34

20 SCW DESIGN REQUIREMENTS 35

20.1 DESIGN DRIVERS 35
20.2 OPERATIONAL FUNCTIONS 37
20.2.1 LEO/MEO DRIVEN FUNCTIONS 37
20.2.2 GEO DRIVEN FUNCTIONS 37
20.2.3 COMMON FUNCTIONS 38
20.3 MANAGEMENT FUNCTIONS 39
20.3.1 SYSTEM OPERATIONS 39
20.3.2 FAULT MANAGEMENT 40
20.3.3 CONFIGURATION MANAGEMENT 40
20.3.4 PERFORMANCE MANAGEMENT 40
20.3.5 SECURITY MANAGEMENT 40
20.4 USER INTERFACE 40
20.5 GENERAL DESIGN REQUIREMENTS 41
20.5.1 SCALABILITY 41
20.5.2 USER SW MAINTENANCE 41
20.5.3 DISTRIBUTED ARCHITECTURE 41
20.5.4 INSTALLATION AND UPGRADE PROCESSES 41
20.5.5 REMOTE PROTOCOLS AND INTERFACES 42
20.5.6 DATA MANAGEMENT 42
20.6 DESIGN STRATEGY 42
20.6.1 OBJECT ORIENTED 42
20.6.2 ADAPTABLE 42
20.6.3 SMALL PIECES 43
20.6.4 DIVORCE USER INTERFACE FROM INSTRUMENTATION 43

1 Introduction

System Control Works (SCW) is a response to the requirements for a flexible, scalable, robust and resilient control system framework.. It is an implementation of a framework that meets these requirements, as well as using recent industry standards to provide the foundation of a system that is inherently scalable, extensible, and easy to deploy and maintain in many differing custom configurations. The main goals of the architecture are:

- ☑ New system configuration without code changes
- ☑ Swapping instrument types without code changes
- ☑ Adding new instruments without changes to existing code
- ☑ Additional functionality available to all installations without multiple code changes
- ☑ Modular architecture to reduce the size of the code base and reduce maintenance costs
- ☑ Comprehensive diagnostic and logging capability
- ☑ Improved scalability to support multiple levels of control

This document is a definition of the architecture and is more concerned with describing the system from the perspective of the user rather than the developer. The user of the system is any of the following:

- The end customer
- The systems engineer designing the system
- The field technician installing the system

Where implementation is described, it is not intended to limit the developers to a single implementation strategy, but rather to illustrate how the architecture can be realized with the languages, tools and protocols available. The actual implementation may improve on these initial implementation proposals.

2 Overview

System Control Works is an object-oriented framework. Careful design of the class hierarchies locates common functionality in base classes, much as libraries provide in a procedural implementation. It is intended that the standard base classes are rich in function, and perform much of the core functionality of the system. Application level classes only need implement that functionality that is specific to the instrument or operation in question. This radically reduces the size of the code base to maintain and the effort involved in adding new instruments and functions to the system.

The framework is built on the industry standard Common Object Request Broker Architecture (CORBA). This provides all the low-level mechanisms for cross platform distributed inter object communications while saving significant implementation time (80%) over the alternative, ASCII messaging (as implemented in the NASA Data Stripper). Once the objects are CORBA-enabled, methods and attributes on remote objects may be accessed as if they were being accessed through a local pointer. To make the objects CORBA enabled their interface must be defined in the CORBA Interface Definition Language (IDL), much as C++ classes are defined, and the object class inheriting the base class generated by the IDL compiler.

In addition, many of the generic services used by the system are supplied with the CORBA implementation.

The overall framework is comprised of components that fall into one of the following categories:

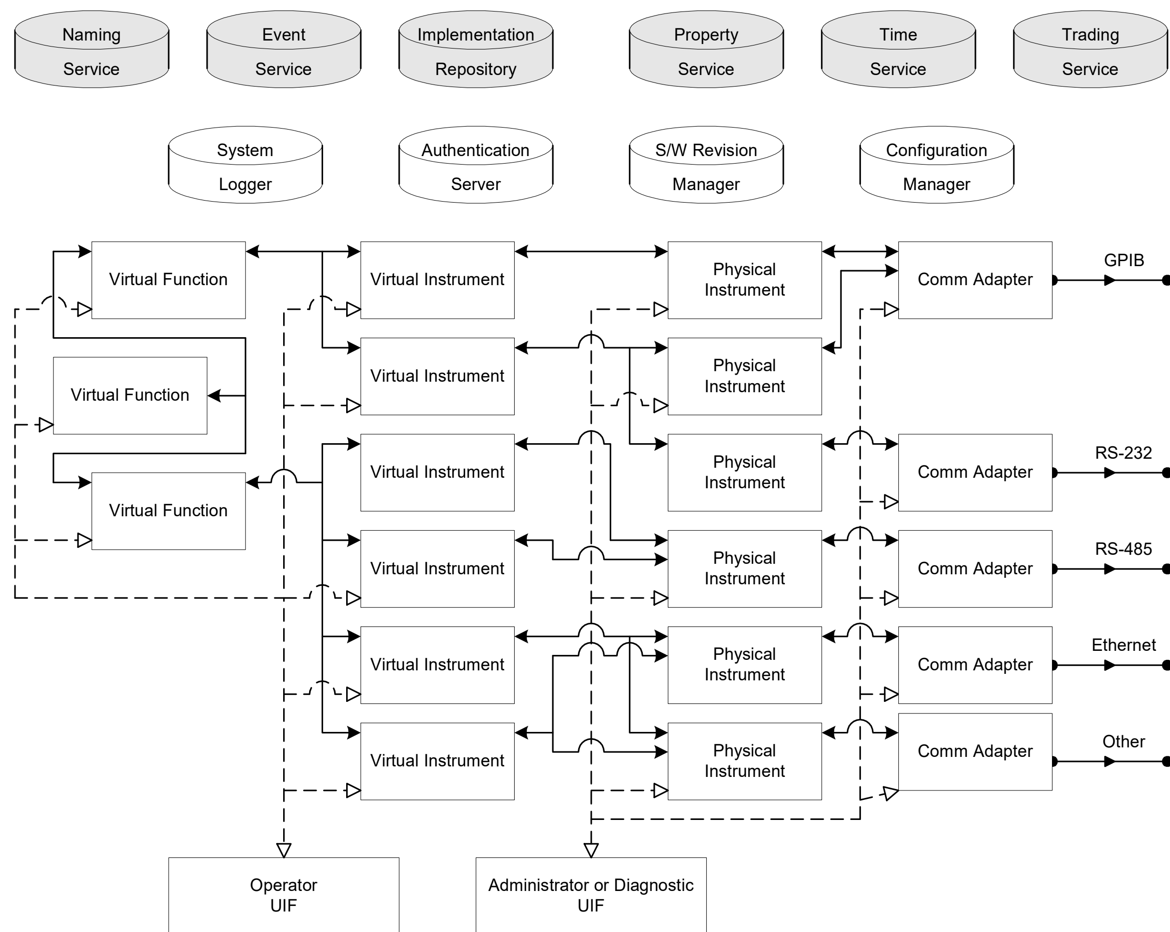
- ❑ Virtual Function Objects
- ❑ Virtual Instrument Objects
- ❑ Physical Instrument Objects

- Communications Adapter Objects

Other special objects provide services to the framework or allow user access to the components comprising the framework. These are:

- User Interface Modules
- Authentication Server
- System Logger
- Configuration Manager
- Revision Management Repository
- System Configuration Tools

The relationships and interconnections between these modules can be represented diagrammatically:



The details of this framework will be described in the succeeding sections.

3 CORBA

At the most basic level, CORBA is a standard for distributed objects. CORBA allows an application to request an operation to be performed by a distributed object and for the results of the operation to be returned back to the application making the request. The application communicates with the distributed object that is actually performing the operation. This is basic client/server functionality, where a client issues a request to a server and the server responds back to the client. Data can pass from the client to the server and is associated with a particular operation on a particular object. Data is then returned to the client in the form of a response.

The advantages of CORBA over other competing technologies are as follows:

- CORBA supports many existing languages. CORBA also supports mixing these languages within a single distributed application. CORBA uses a standard interface definition language (IDL) to define the distributed objects. The same IDL is used in both language mappings. Since different pieces of the entire station control software will be written in different languages, primarily C++ and Java, the use of CORBA IDL ensures that all modules, regardless of implementation language, will be compatible.
- CORBA supports both distribution and Object Orientation.
- CORBA is an industry standard. This creates competition among vendors and ensures that quality implementations exist. The use of the CORBA standard also provides the developer with a certain degree of portability between implementations. Several ORBs exist that are available as open source, and one of these will be used for the initial implementation. If additional features are required that are not available in the open source ORB it will be possible to switch to a commercial ORB without requiring extensive code changes. If care is taken to adhere closely to the language mappings, no code changes will be necessary.
- CORBA provides a high degree of interoperability. This insures that distributed objects built on top of different CORBA products can communicate. Different ORBs may be used for different pieces of the system. For example, new instruments may be developed, such as the 38XX ACU, which contain an embedded operating system. This operating system may require the use of a different ORB from the one used by the rest of the software. However, the interoperability features of CORBA guarantee compatibility.

3.1 CORBA Services

The Object Management Group (OMG) has defined a number of services to provide additional capabilities to basic CORBA mechanisms. These are as follows:

- Event - a simple messaging mechanism for decoupling objects
- Naming - an object directory that allows objects to be located by a name
- Trading - another object directory that uses object properties in searches
- Notification - a more sophisticated messaging system, that supports guaranteed delivery and event filtering
- Life Cycle - provides support for the object lifecycle
- Property - allows values (properties) to be associated with an object at runtime
- Collection - supports the manipulation and management of collections of objects
- Concurrency - a mechanism for managing the concurrent access to system resources
- Relationship - allows networks of objects to be created and navigated.
- Time - a service for providing consistent time across a distributed system.

The services of particular relevance to System Control Works are described below.

3.1.1 Naming Service

The OMG Naming Service provides a mapping from names to object references. Given the name, the service returns the object reference stored under that name. This is much the same as when a DNS server maps system names to IP addresses. Objects register with the name server and thus become public. The names used are arbitrary but must be unique.

The Naming Service maps names to object references. A name-to-reference association is called a *name binding*. The same object reference can be stored several times under different names, but each name identifies exactly one reference. A *naming context* is an object the stores name bindings. The naming context implements a table that maps names to object references. A name in the table can denote either an object reference to an application object or another context object in the Naming Service. This means that, like a file system, contexts can be connected to form hierarchies: contexts correspond to directories that store names either to directories (other contexts) or to files (application objects).

This hierarchical naming structure lends itself very well to the clustering requirements of the design architecture. Similar groupings of instruments can be assembled and given the same names, but will belong to different naming contexts, just as files in a file system can have the same name once they are in different directories. As an example of implementation, a naming hierarchy such as *customer/location/station/node/device* could be used. This would create a unique naming convention for all SCW based systems deployed.

Objects seeking to establish a connection to a published in object issue a name resolution request to the name server and are returned a unique reference to the object. The object reference acts as the equivalent of a pointer.

3.1.2 Trading Service

The Naming Service allows a client to locate an object and acquire its reference with the use of a symbolic name. This has been likened to the white pages of a telephone directory. The Trading Service on the other hand allows a client to obtain an object reference without a symbolic name but based on the services offered by the objects.

The trader stores advertisements for services. A stored advertisement is known as a *service offer*. A service offer contains a description of the service as well as an object reference to the object that provides the service. The act of placing an advertisement is known as an *export* operation. The program or person who places the advertisement is called the *exporter*. The description of the service inside a service offer is provided by a number of name / value pairs called properties. The object reference inside a service offer denotes an object that provides the advertised service. That object is known as the *service provider*.

These capabilities of the Trading Service would be used to allow object discovery, particularly for use by the user Interface modules.

3.1.3 Event Service

Basic CORBA uses a synchronous request model. The CORBA Event Service provides asynchronous communications. CORBA objects are either suppliers or consumers of events. The Event Service connects suppliers and consumers through the mechanism of event channels. Suppliers can push data into the event channel or wait for it to be pulled. Similarly, consumers can wait for data to be pushed to them from the event channel or pull it from the event channel. Event channels can support multiple suppliers and consumers, and suppliers can be both push and pull suppliers and consumers can be both push and pull consumers, on the same event channel. This mechanism supports a very flexible configuration of communication between CORBA objects. The same data can be pushed from the supplier and pushed onward to a push consumer and buffered for data collection by a pull consumer.

Applications for the Event Service within the SCW architecture include the interface between the communications adapters and the physical instruments, the interface between the UIF modules and the virtual instruments and virtual functions displayed, and the statistics module and the various modules providing repetitive data and status.

Most of these applications will be implemented with push suppliers and push consumers. Other configurations may be used as appropriate.

3.1.4 Notification Service

The CORBA Notification Service extends the Event Service. The main extensions are:

- Event filtering -- which allows data pushed by supplier on an event channel to be filtered before it is delivered to the consumer. This simplifies applications where several consumers are monitoring the same supplier but require a different subset of the data being sent.
- Quality of service -- which allows tuning of event channel to optimize reliability and performance characteristics.

3.1.5 Time Service

The CORBA Time Service provides several time-related facilities:

- It enables the user to obtain current time together with an error estimate associated with it.
- It ascertains the order in which “events” occurred.
- It generates time-based events based on timers and alarms.
- It computes the interval between two events.

The CORBA Time Service consists of two services and defines two service interfaces:

- The Time Service manages Universal Time Objects (UTOs) and Time Interval Objects (TIOs), and is represented by the *TimeService* interface.
- The Timer Event Service manages Timer Event Handler objects, and is represented by the *TimerEventService* interface.

3.1.6 Property Service

The CORBA Property Service provides the ability to dynamically associate named values with objects outside the static IDL-type system. This allows client applications to manipulate data in CORBA objects without being built with the IDL mappings for these objects. This is exactly the case with the SCW user interface modules, which are written without prior knowledge of the function and instrument objects comprising the system.

It defines operations to create and manipulate sets of name-value pairs or name-value-mode tuples. The names are simple OMG IDL strings. The values are OMG IDL *an*ys. The use of type *any* is significant in that it allows a property service implementation to deal with any value that can be represented in the OMG IDL-type system. The modes are similar to those defined in the *Interface Repository AttributeDef* interface.

It is designed to be a basic building block, yet robust enough to be applicable for a broad set of applications, and provides “batch” operations to deal with sets of properties as a whole. The use of “batch” operations is significant in that the systems and network management (SNMP, CMIP, ...) communities have proven such a need when dealing with “attribute” manipulation in a distributed environment.

It provides exceptions such that *PropertySet* implementors may exercise control of (or apply constraints to) the names and types of properties associated with an object, similar in nature to the control one would have with CORBA attributes. Thus, the user interface can be restricted as to values that will be accepted, and attempts to update parameters without permission can generate exceptions.

It allows *PropertySet* implementors to restrict modification, addition or deletion of properties (readonly, fixed) similar in nature to the restrictions one would have with CORBA attributes.

It provides client access and control of constraints and property modes. This could be used to allow an administrative console to override limits coded into the objects or defined in the system configuration.

4 System Control Works Components

SCW objects share the following attributes:

- Self-contained, named objects.
- Self-configuring.
- Self-referencing.
- Similar objects export a standard interface.
- Fully distributable on disparate platforms.

A system can be built from as many or few SCW objects as required. Since all objects are self-referencing and self-configuring, the entire system will self-configure once the list of objects and all the interconnections are defined. Specific support for individual instruments need not and should not be coded in other objects. If variations exist and must be handled, the objects can exchange capabilities and requirements and effectively negotiate a working interface. Due to the fine partitioning of the system such negotiation should be minimal and in respect to very narrow feature sets.

4.1 Virtual Function Objects

Virtual functions implement defined functions within the system. These would include the functionality implemented in the current system by such modules as SCHEDULE, TRACK, UACTASK and various test modules.

Some virtual functions are persistent and are present as long as the system is active. SCHEDULE is such a persistent function. Others are transient, and are instantiated only when required. System tests would fall into this category.

Virtual functions use the services provided by other virtual functions and virtual instruments. This isolates the functions from the specifics of the actual devices used in the installation.

Virtual functions may vary from the simplistic to the highly complex, and may connect to just a few other virtual devices or many. Where possible, highly complex operations should be divided between a number of virtual functions with a master controlling function over all. This highly modular, compartmentalized approach will reduce the maintenance demands of the system.

There are three special virtual function objects that are derived from the base virtual function classes, but have important roles in the system.

4.1.1 Node Controller Virtual Function

The node controller virtual function enables the multiple levels of system control. Using the configuration information and discovery methods in the system it builds an image of the system and exports a control interface for use by a remote master system controller. The base node controller class implements the local system control, while derived classes will implement the protocol used by the remote controller, such as SNMP, HTTP, or CORBA.

In addition derived node controller objects may store UIF profiles so that remote controller GUI engines can open the node and see a full interface without having to build a profile at the UIF end of the connection.

The node controller also serves as the arbiter of point-of-control, and manages access to the resource sets required for various system operations.

4.1.2 Script Engine Virtual Function

The script engine virtual function is a module that provides an interface between user or vendor supplied scripts and the system. Many operations can more simply be developed as scripts.

The script language will be an industry standard. Due to its object-oriented nature, Python appears to be a good candidate.

4.1.3 Statistics Virtual Function

The statistics virtual function is used in systems that require repetitive collection of data and status statistics. Data structures will be defined for each module that records statistics. The Statistics Virtual Function will use the CORBA Event Service to subscribe to the statistics data. Part of the subscription process will be the specification of the update interval.

The definition of the data structures to the used for statistics gathering needs to be in separate header files, rather than in the class definitions for module gathering the statistics or for the Statistics Virtual Function. This will insulate the Statistics Virtual Function from the implementation of the various modules being monitored, and the modules from the Statistics Virtual Function. These data structures could well be defined as SCW specific MIBs, which would make them suitable for SNMP monitoring.

Once the statistics have been collected the data can be saved to files or reformatted and forwarded to an external module or process. Different derived classes of the Statistics Virtual Function can be implemented inheriting from a base Statistics Virtual Function. The base class would implement all the data collection methods and the interfaces to the modules that gather statistics, while the derived classes would implement the disposition of the data once collected.

4.2 Virtual Instrument Objects

Virtual instruments objects provide a standardized functional interface for the various types of instruments used to build a complete station control system. The specific instruments used may depend upon performance, price, availability, or customer preference. The functionality, however, is similar.

Virtual instruments implement the complete functionality for a generic system component and export a single interface. Some functionality may be optional, so the virtual instrument will maintain a capability profile. Virtual functions will thus be able to tailor themselves accordingly.

Virtual instruments map the generic interface to the specifics of the physical instrument objects. Some virtual instruments, such as the X-Band receiver, are implemented by several physical instruments. Others use just a subset of the capabilities of the physical instrument. Yet others are mapped one-to-one to physical instruments.

Virtual instruments serve as a reductive adaptation layer, in that they reduce the instrument functionality to its simplest form. The command set exported by a virtual instrument will most likely be smaller but more ideal than the physical device it represents. This also serves to reduce complexity in the system by separating out unrelated capabilities of the physical device for use by a different virtual instrument. It also reduces the variations in the abstract functional layers of the system, thus minimizing the effects of adding or changing generic functionality and making it simpler to share such improvements among differing installations.

Where there are several combinations of instrument personalities that may be used to implement the functionality of a virtual instrument, the virtual instrument object will use the CORBA _narrow() method to identify the exact type of the subordinate module. A smart virtual instrument will be able to tailor itself to make appropriate use of the subordinate modules to implement the required functions and features.

4.3 Physical Instrument Objects

Physical instrument objects encapsulate the capabilities of a specific device. Thus, there is a one-to-one mapping between a physical instrument object and the device it controls. At the same time it exports a standard interface to the virtual instrument objects, according to functional personalities. Some capabilities of the standard interface may be fully supported while others may be emulated within the physical instrument.

The physical instrument object thus serves as an additive adaptation layer. For devices with fewer capabilities than the standard interface exports, the physical instrument object implements a safe buffer for these calls. Virtual instruments therefore need less negotiation to connect safely to the physical instrument. In addition, where devices of similar function use different metrics for represent data the physical instrument will provide conversion facilities to a standard metric used by the interface. An example of this would be that of recorders, where tape position may be represented as time, inches, centimetres, or some other arbitrary unit.

A major purpose of the physical instrument object is to decouple the actual device from the rest of the system. A device may be limited as to I/O bandwidth, or particularly state-conscious in such a manner as to limit certain operations at some times. The physical instrument object will provide unrestricted, unconstrained access to the device's functions to the rest of the system, while enforcing whatever system and device specific limitations must be placed on these functions.

Physical instrument objects, like the devices they embody, may be comprised of multiple functional personalities. Some of these are related, while others, such as the discrete I/O contacts on the SA3860 ACU, can be used for completely unrelated operations. To support this the physical instrument can export multiple personality interfaces, each of which can be used by a different virtual instrument.

Physical instrument objects communicate with the devices they embody via virtual circuits provided by communications adapters. All the physical instrument needs to know is the name of the communications adapter to use, which it gets from the configuration manager.

4.4 Communications Adapter Objects

Communications adapter objects correspond to I/O handlers in the present software. Each adapter manages one group of connections. These will typically be an IEEE488 bus, a multi-dropped serial link using RS-485, a single point-to-point serial connection, or some other medium or protocol. Some devices may be implemented as PCI cards in a server PC.

The interface exported by a communications adapter is common to all transport types. This makes it transparent to the physical instrument as to which bus or protocol it is using. The communications adapter obtains the list of connected devices from the configuration manager and matches device names to protocol addresses. It then creates a unique virtual circuit object that is used by the physical instrument object to communicate with the device.

In the case of bus connected devices, a custom communications adapter must be written to translate I/O commands from the physical instrument object to device driver invocations.

4.5 User Interface Modules

User interface modules can access exported data and controls in all components: virtual functions, virtual instruments, physical instruments and communications adapters. Access is subject to authentication.

The interface modules themselves have no specific knowledge of the significance of various controls and data values. They are just user interface engines. Each modules exports data and controls with names and properties. The user interface discovers the modules and selects the data and controls to be displayed, subject to authentication permission. The selected layout is saved in a profile that will be loaded on login.

While this mechanism well supports the existing UIF implementation, it is more extensive. Debugging and high level monitoring profiles can be created that provide an interface directly to the physical instrument or communications adapter.

Profiles thus generated may be saved in node controller virtual functions, so that by accessing the node controller an administrator will be able to monitor or control a system without having to generate a specific profile for it. This is of special benefit where there are many stations comprising a larger system requiring centralized monitoring and control.

4.6 Authentication Server

The authentication server uses a secure connection (SSL) to validate client access to various objects within the system. Each object supports two levels of access: monitor and control. These correspond to read only and read/write file access. When a client attempts to access an object it will make a request and receive one of the following: full access, monitor access, or no access. Due to the requirements of multiple points of control a client's full access may be revoked at any time and reduced to monitor access.

The role of the Authentication Server is discussed in more detail in the section Point of Control Arbitration.

4.7 System Logger

Built into the base classes inherited by all objects are logging methods that filter log calls by type and severity. Exported methods set the type mask and levels. Log calls that do not pass the filter requirements are effectively ignored with minimal system overhead. Calls that pass are sent to the system logger and are saved in files. Within the logger, messages are directed to various files based on the source module, log type, and log level, based on a set of filters. Messages are logged to as many files as the filters permit. (This approach is similar to the Unix *syslog* facility.)

The logging method implemented within the System Logger is essentially one of a filtered push. For systems with limited data and status monitoring requirements, the system logger may be adequate. If the filters are correctly written, standard log output from various modules will be available in separate files. The nature of the logged data and the rate at which it is logged would have to be coded into the individual modules. Exported methods will allow system control of the frequency of logged data.

For more sophisticated logging of data and status it will be necessary to use the Statistics Virtual Function and the event service.

4.8 Configuration Manager

The configuration manager acts as a repository for the system configuration, as well as multiple resource configurations. The system configuration is generated by a system configuration tool based on information provided by the object.

When objects initialize, they interrogate the configuration manager for names of connected objects, and any other startup parameters presented to the configuration tool by the object.

The configuration manager is also responsible for creating the initial objects required by the system. The CORBA framework differentiates between objects and the servants that incarnate them. Objects can be created and published either by the Naming Service or by the Trading Service without the processes that incarnate the objects existing. Activation of the objects and incarnation of the servant processes can be deferred until clients connect to them.

The configuration manager will also implement a watchdog function that pings static objects to make sure they are still alive. It will also keep track of transient objects and ping them during their lifetime to verify their existence.

What is the update path to modify the configurations from GUI, SNMP or other CORBA objects?

4.9 Revision Management Repository

All objects must contain revision information, both for the object implementation itself, and any base classes it inherits. Thus, an object's actual version is the sum – by string concatenation – of all the versions in its inheritance tree. Since much of the implementation is inherited from base classes compatibility much be ensured at each class level. To expose this information all objects will register with the revision management repository, which will simplify tracking of the revision level of the whole system.

4.10 System Configuration Tool

The system configuration is concerned with three areas:

1. The list of objects comprising the whole installation.
2. The static interconnections between the objects.
3. The default configuration of the individual objects.

The system configuration tool is the means by which this information is generated. (At the outset, this may simply be a text editor and a sharp mind. Later it will evolve into a fancy GUI drag/drop tool.)

4.10.1 List of Objects

The list of objects simply defines which objects will be instantiated by the system. The list will define the object class and instance name. The instance name must be unique to the system and is registered with the name server. This name can be used to resolve a reference to the object, wherever it may reside.

4.10.2 Static Interconnections

Each object class may have a number of attachment points for connection to other devices. The self-referencing feature of the classes is used to expose these to the configuration tool. Objects with no attachment points can only support dynamic connections, and are typically server objects such as the configuration manager or logger. Where objects have only one attachment point, it is optional.

Each attachment point serves as on end of a point to point connection. Static interconnections are expressed in terms of object name / attachment point pairs. When objects are initialized, they resolve the references for the necessary static connections using the configuration manager to retrieve the interconnection details, the name server to resolve the object name, and the object itself to resolve the attachment point.

4.10.3 Object Defaults

Each object class may have a number of configurable parameters. Some may be static and fixed for the installation. Others are dynamically configurable but require installation specific defaults. The self-referencing feature of the classes is used to expose these to the configuration tool. The tool then uses the validity checking features exposed by the object to allow the user to configure the parameters.

5 Implementing Components

Components are implemented in dynamic linked libraries: DLLs under Windows NT & 2000, and shared objects under Unix. Within the DLL are all the objects related to creation, operation, and management of the component. Specifically, this requires the development of three distinct, related classes:

- ❑ The object factory class
- ❑ The CORBA object servant (implementation) class
- ❑ The object management class

5.1 The Object Factory Class

The role of the object factory class is described later in more detail in the section “Operational Scenarios”. Its basic responsibility is to incarnate CORBA objects of the appropriate type. Some factories will be implemented to create multiple related object types, while other will create just one. Where objects must exist as singletons, the factory will enforce this.

5.2 The CORBA Object Servant Class

The CORBA object servant class provides the implementation of the object function. This is the piece that actually does the work. Servant classes are as diverse as there are components.

One feature that all servants share is the parameter list. Parameter lists provide a mechanism for modules exporting data and controls to objects that are not directly configured to interface to them. The prime example is that of the User I/F modules, which just locate objects by name, and add GUI interface objects to them.

5.3 The Object Management Class

The object management class is a descriptive class for the object servant.

Some modules – primarily the Node Controller virtual function and the System Configuration tool – need to know how the servant interconnects with other modules, without needing to know anything about the internals. The object management class documents these in such a way that the node controller can build a matrix of objects representing the entire aggregation of modules in the system.

6 Module interconnections

There are two types of interconnection between modules in the SCW system: static and dynamic.

6.1 Static Interconnections

Static interconnections are designed into the system when it is configured, and reflect the hard wiring of the various physical devices in the system. Object references required for static interconnections are obtained by querying the configuration manager for the name of the connected object and the name server for its object reference. Examples of static interconnections are:

- physical instrument objects and communications adapters,
- virtual instrument objects and physical instrument objects,
- virtual instrument objects (such as switches and multiplexers) and the virtual instrument objects they switch.

6.2 Dynamic Interconnections

Dynamic interconnections are created while the system is running. Object references required for dynamic interconnections are obtained by querying other modules that control which objects are appropriate for the context. Examples of dynamic interconnections are:

- virtual function objects and virtual instrument objects, such as for test operations,
- the node controller virtual function object and virtual function objects, virtual instrument objects, physical instrument objects, or communications adapters, for aggregated control and monitoring of the node,
- virtual instrument objects and other virtual instrument objects, where settings need to be shared,
- user interface modules and virtual function objects, virtual instrument objects, physical instrument objects, or communications adapters, depending on the nature of the user interface modules.

6.3 Example of Interconnections

An example of both static interconnections and dynamic interconnections would be the implementation of the exciter redundancy switch, as used by the uplink subsystem. The following diagram illustrates this.

The connections between the exciter redundancy switch and the exciter virtual instrument objects are static, as are the connections to the discrete I/O physical instrument objects. The connection between the uplink virtual function object and the exciter redundancy switch is also static, although it would not be resolved until the uplink virtual function object is activated. Nevertheless, only one exciter redundancy switch would be used for the uplink function. (If a complex system were designed with multiple uplink subsystems and it may be necessary to use dynamic connections between multiple uplink virtual function objects and multiple exciter redundancy switches.)

The connection between the uplink virtual function object and the uplink exciter virtual instrument object is dynamic, and is obtained as an object reference by querying the exciter redundancy switch.

7 Self-referencing Modules

All modules maintain a list of exported parameters. Parameters are named, typed, and where appropriate, include validation information. It is not expected that parameter types are complex data types. Statistics would be recorded in data structures but these are not accessed by the normal parameter interface methods.

Parameter data types would include integers, floating point and text string. CORBA supports integers up to 64 bits and single (32-bit) or double (64-bit to) precision floating point data. Data validation for integers and floating point data would be a range (minimum and maximum). The validation for text strings would be length, and, optionally, allowed characters.

Enumerated data types are supported in CORBA, but these must be defined in IDL. IDL enumerations cannot be safely mapped to integers. Therefore, it is not suitable to use the enumerations within the self-referencing features of modules.

If user interface modules wish to gain access to the IDL definitions of modules will have to access the CORBA interface repository. This introduces dynamic interfaces to the architecture, which is not planned at present. However, more research will be done into the dynamic interface model to see if this can easily be implemented for user the interfaces. If it can it may well prove to be an alternative to a custom self-referencing feature of SCW modules.

The two uses of the self-referencing capability are the user interface modules, which by definition, do not have specific knowledge of the internals or the external characteristics of the modules whose data they will display, and the system configuration tool which uses the self referencing data to configure defaults for the modules for initialization purposes.

Work has already been done on flexible configuration mechanisms in the ASCII config efforts for the NASA 11m enhancements, and this will provide a foundation for this feature.

8 Point of Control Arbitration

An essential issue in the SCW architecture is the need for a flexible part of control scheme. The potential exists for multiple points of control vying for primacy. These points of control may be user interface modules, with a human operator logged in with some permissions level, an external control source using SNMP or some other protocol to control station, or automated operations based on a saved schedule. Also, access to the modules may be required simply for monitoring and not for control.

The basic principle of point of control arbitration for the system will be that only one point of control will have read/write access to a module, while multiple points may have read-only access. Read/write access may be preempted when a high-priority part of control is present.

It is also desirable to establish point of control to an aggregate of modules. This would be done by gaining control of a node controller virtual function that would in turn to establish itself as the part of control for each module in the aggregation.

Possible sources vying for control of the system are as follows:

- Multiple, possibly overlapping, scheduled events (passes)
- System operator performing non-critical monitoring and test operations
- System supervisor performing critical system updates or operations.

In addition to the varying priority level of these conflicting demands, the resource sets required by each may be different. For this reason the node controller will assemble resource lists corresponding to the system components required to perform the requested operation and grant access if no equal or higher priority point-of-control is using them. If a point-of-control is preempted by a higher priority request control of all resources in its resource list will be revoked. Resource lists are thus treated as atomic, to ensure deadlocks cannot occur.

A point-of-control CORBA interface must be defined that provides a callback mechanism to provide a grant of control to the requesting module. The same callback will be used to revoke control if a higher priority point-of-control appears.

8.1 Access Control

Access control for users and remote points of control will be by user ID and password. These will be translated by the authentication server into an access control code with an associated priority level. The priority level will be used to arbitrate between multiple points of control. Where more than one point are vying for control and share the same priority level the first one will gain control.

8.2 Secure Connections

To support remote points of control, the communication involving passwords and access control codes must take place over secure connections. Many CORBA implementations have support for the SSL protocol that provides this.

9 Fault Logging and Recovery

Extensive use of `try/catch` blocks within the code is necessary since the CORBA C++ mapping uses exceptions for error handling. All caught exceptions will be reported to the system logger.

The watchdog thread within the Configuration Manager will monitor all modules and detect when they are lost. These cases will also be logged.

If there is a Node Controller virtual function, it will be notified of the health of the modules. (The exact protocol for this remains to be defined.)

10 System Initialization

10.1 System Startup

10.1.1 SCW Universal Manager

The SCW universal manager is needed to support Discretionary Access Control under Windows NT. NT applications cannot be configured to gain access to system resources restricted to the user. To limit access to sensitive to programmatic control only requires a service that will act as a manager for the system.

The manager will be written to provide a minimum of essential services, and be version independent. The remainder of the code will be implemented in DLLs. These will be found in version specific directories as shown in the section “Filesystem Hierarchy” below.

The manager will determine which version of the software to run from a key file in its startup directory. This will be used to locate the version in the version dependent portion of the tree.

Default versions of the DLLs and executables may be placed in the universal *bin* directory. The manager will always search the version specific tree for files before loading default code.

10.1.2 Configuration Manager

The Configuration Manager is responsible for starting all the initial processes required for the system, and activating the necessary objects. Object factories will be implemented, and daemon processes will be required on all servers where objects will be incarnated. If an object is to be incarnated on a remote system, a message must be sent to the remote object factory to create the object servant.

Initiation of the system must be possible automatically or manually. Scenarios for startup are:

- Fully automatic, when the server is powered-up
- As soon as the control account is logged-in
- Manually, from a desktop icon or menu item.

The configuration manager will use the system configuration file to determine which modules to load, and what initial parameters to pass to them when requested. The goal is that configuration files will not need to be tied to a particular version of the software and may be placed in the universal *cfg* directory. All configuration files are marked with the version that created them. If version dependencies are introduced, conversion tools must be supplied. Converted configuration files must be placed in the version specific cfg directory. The configuration manager will first search the version specific *cfg* directory for its configuration, and then the universal *cfg* directory.

10.2 System Shutdown

The manager will accept a shutdown command to terminate the software. There will be two shutdown modes:

- **Halt** will send shutdown messages to all modules, and then terminate the manager service.
- **Restart** will send shutdown messages to all modules, unload all manager DLLs, and then reread the key file before reloading the DLLs and starting the required system modules.

10.3 Revision Management

10.4 Version Switching

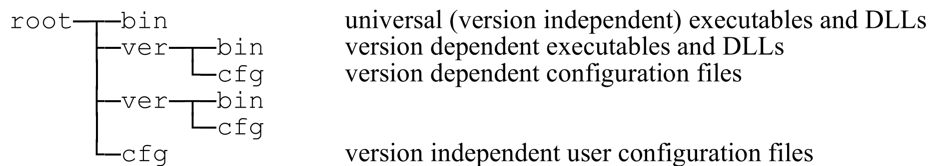
As described above, the SCW Universal Manager will use a key file to locate the version of the DLLs and executables to run. All versions used by the site may be preserved on the server. To switch to a new or previous version of the software just the key file will need to be altered and the manager issued a restart command, as noted above.

Installation of a new version of the code will involve the following steps:

1. Create the version specific directory tree, and copy the new binaries and configuration files into it.
2. Check and convert any incompatible configuration files currently in use.
3. Update the key file to point to the new version.

10.5 Filesystem Hierarchy

The software files will be laid out under a common root directory as follows:



The log files will be saved in a separate hierarchy and laid out under a common root directory as follows. This will allow log files to be saved to a separate, possibly remote, filesystem.

11 SCW class hierarchy

11.1 SCW Base Class

The SCW base class is an abstract class from which all SCW modules and components descend. Methods and attributes implemented in the SCW base class are as follows:

- Log level and log category mask
- Log methods
- Static name server lookup methods
- Name server registration methods
- Class name and revision attributes
- Object name attribute

11.2 SCW Component Class

The SCW component class is an abstract class derived from the SCW base class. It adds discoverable parameters. Parameters are defined by data structures that include the parameter name, its value, and its type and validation rules. Methods and attributes implemented in the SCW components class are as follows:

- Parameter list
- Parameter access so methods

11.3 Instrument Personality Classes

Instrument personality classes are mix-in classes that primarily define the interface for the various devices used in the system. Instrument personalities include:

- Signal Generator
- ACU
- Receiver
- Recorder
- Switch
- Discrete I/O
- Power Meter
- etc

These classes are combined with the SCW component class to create virtual instruments. By definition, only one personality mix-in would be included in the inheritance of a virtual instrument, since they are designed to embody a single instrument interface.

These classes are also combined with the SCW physical instrument class to create physical instruments. Multiple personality mix-ins may be included in the inheritance of a physical instrument, depending on the capabilities provided by the device.

12 Virtual Instrument Class

The physical instrument class inherits from the SCW component class, and the instrument personality mix-in class for the instrument to be virtualized.

13 Physical Instrument Class

The physical instrument class inherits from the SCW component class, and as many instrument personality mix-in classes as necessary to match the capabilities of the device. Functions performed by the physical instrument class are as follows:

- Command translation from SCW instrument methods and metrics to device command strings
- Command emulator for SCW instrument methods not implemented by device
- Status cache for instrument settings
- Polls instrument at rate suitable to device to refresh status cache
- Monitors device to detect instrument disconnect or failure

14 Communications Adapter

The communications adapter is implemented as a base class from which are derived specific communications adapters for the various protocols used to communicate with the devices. At present communications adapters are planned for GPIB, serial devices, and TCP/IP connected devices.

The Communications Adapter will encapsulate the hardware differences in each of the busses and present a consistent interface to the Physical Instruments. This will be achieved by two means: the use of base class methods and a virtual circuit class.

The virtual circuit class will handle the multiplexing of communications in multi-point scenarios and simply pass the data through in point to point scenarios.

14.1 Base Communications Adapter Class

The communications adapter class is derived from the SCW component class.

14.1.1 Module Function

The base communications adapter class implements the functionality required to manage virtual circuits requested by the physical instrument objects. Virtual circuits are identified by an arbitrary address string that is converted by the derived adapter class into a meaningful protocol address.

The standard interface between the communications adapter and the physical instrument supports the following operations:

- Open bus
- Open virtual circuit – returns virtual circuit handle
- Close virtual circuit
- Reset all virtual circuits
- Send data to device
- Receive data from device
- Report loss of connection to device
- Report reconnection to device

All operations are synchronous and return immediately.

The Communication adapter provides a private, standardized interface to the virtual circuit objects. The virtual circuits use these to accomplish their tasks. These mask the hardware and configuration differences. This interface includes:

- Send data to device
- Receive data from device
- Report loss of connection to device
- Report reconnection to device

14.1.2 Properties in System Configuration

- None

14.1.3 Embedded Objects

- Virtual circuit object.

14.1.3.1 Virtual Circuit Class

The virtual circuit class is a helper used the mask the differences in communication to the different types of Communication adapters and to encapsulate each data stream.

The interface between the virtual circuit and the physical instrument supports the following operations:

- Close virtual circuit
- Reset virtual circuit
- Send data to circuit
- Send device health query string and health indication comparison string
- Receive data from circuit
- Report loss of connection to device
- Report reconnection to device

All operations are synchronous and return immediately. Send data must be buffered in the communications adapter if the I/O medium cannot accept it unless the connection is broken but the virtual circuit is not yet invalidated, in which case the data is discarded while returning an error code to the Physical Instrument. Receive data must be buffered in the physical instrument if it cannot be processed immediately.

14.1.4 Methodology

Physical instrument objects will fetch the name of the Comm Adapter they should use from the Configuration Manager. This is described in the section "Operational Scenarios: System Startup: Physical Instrument".

The physical instrument object will invoke the *OpenVirtualCircuit* method on the comm adapter with the specific connection address string and a selection for push or pull. The comm adapter will parse and compare the address to its known connections and make the necessary connection. If the operation is unsuccessful, it will return a standardized error code. Otherwise it will instantiate a VirtualCircuit object with necessary control information to correlate it to this connection, and return the CORBA object reference to the virtual circuit object to the physical instrument object. The virtual circuit object should operate in it's own thread.

Virtual circuits are exclusive, one-to-one connections. If a circuit already exists to a specific device address, subsequent attempts to open a virtual circuit to it will fail. If the same module attempts to reopen the virtual circuit (perhaps following a software crash) the circuit will flush its buffers and be returned to the caller. Such a reopen will not cause any interaction with the device.

To send data, the physical instrument object will invoke the SendData method on the virtual circuit object, passing a buffer containing the data to send. The virtual circuit object, in turn, will make the necessary call in to the comm adapter to send it to the specified hardware. The virtual circuit object will buffer data for its own device.

The comm adapter will receive indications of inbound data from the hardware using an asynchronous notification method (for example, an SRQ handler for GPIB). It will read the data from the channel and place it into a queue for the appropriate virtual circuit object. Then it will trip a semaphore notifying the virtual circuit object of the data's presence. If a PUSH circuit was selected, the virtual circuit object will call directly into the physical instrument object to place the data. Otherwise the virtual circuit object will retain the data until a PULL request is made, when it will return a single message of data and a return code indicating whether more remains.

It is also desirable to implement a monitoring capability into the comm adapter, so an engineer can observe the traffic over a virtual circuit. At times, it may also be necessary to take over the connection for diagnostic purposes. A mechanism is implemented within the comm adapter to support these features. A "sniffer" object can be registered with the comm adapter for a given virtual circuit that will be copied on all data passing through. In addition, the sniffer will be able to hook the virtual circuit and perform full I/O on it. This will appear to an attached physical instrument object as a disconnect event. When the hook is released the physical instrument object will receive a reconnect event.

14.2 GPIB Communications Adapter Class

14.2.1 Module Function

The GPIB communications adapter class is derived from the base communications adapter class, and adds the GPIB protocol specific functionality:

- Mapping virtual circuits to GPIB addresses
- Implementing asynchronous GPIB messaging
- GPIB bus monitoring

14.2.2 Properties in System Configuration

- ❑ GPIB bus number

14.3 Serial Communications Adapter Class

14.3.1 Module Function

The serial communications adapter class is derived from the base communications adapter class, and adds the serial protocol specific functionality:

- Point-to-point RS-232 connection
- Serial link monitoring, if supported in hardware

Many network attached serial concentrators use a TCP/IP port connection to transfer data between the host and the concentrator. Some of these also provide a serial port device driver interface for the host O/S. If such a device driver is available the ports would be supported by the standard serial communications adapter.

14.3.2 Properties in System Configuration

- ❑ Port specification, bit rate and framing.

14.4 TCP/IP Communications Adapter Class

14.4.1 Module Function

The TCP/IP communications adapter class is derived from the base communications adapter class, and adds the TCP/IP protocol specific functionality:

- Mapping virtual circuits to socket connections
- TCP connection monitoring

As noted above, many network attached serial concentrators use a TCP/IP port connection to transfer data between the host and the concentrator, and not all have serial port device drivers available for the SCW target host O/S. Of the remainder some use a single TCP/IP port with a wrapper protocol for the data. Others map each serial port to a different TCP/IP port.

Serial ports on concentrators using a single TCP port for all serial ports will require a special comm adapter that will implement the wrapper protocol, and port setup protocol.

Serial ports on concentrators using a separate TCP port for each serial port might be able to use the standard TCP/IP comm adapter, but if so, they will have to have their line speed and framing configured externally to the system. Also, the address specified in the configuration will have to be the TCP/IP address of the concentrator and port number mapped to the serial port. Alternatively, a special comm adapter can be

derived from the standard TCP/IP comm adapter that translates the port names and settings into the correct TCP/IP address and configuration protocol.

14.4.2 Properties in System Configuration

- IP address and possibly port number

15 Operational Scenarios

The following scenarios outline the key operations within the system, and will be used as the basis of use cases for elaborating the system design. This list of scenarios will grow as the system design is refined and the scope of the design is widened to detail all functionality required of the system.

15.1 System Startup

The System Startup scenario describes what happens when the system is initiated. This may be on power-up, login or program invocation. If the system is spread over multiple servers, the slave SCW daemons must be running before the system can be initialized.

15.1.1 Main server

The main server is the controlling system for the station. Some modules may exist on slave servers. Slave servers do not have a copy of the system configuration. The following pseudocode describes how the main server starts the modules required, based on the configuration file settings.

```
(Start SCW service/daemon – presence of configuration file specifies main server operation)

Register with name server

Open configuration file <SYSTEM> section

Iterate through module list. For each module:

If module is local

    If module does not require a factory

        Start module executable with NAME parameter

    Else

        If factory process [PROCESS_GROUP] does not exist

            Start factory process [PROCESS_GROUP]

        Start module via factory process

Else

    Send creation command to remote slave server
```

The following XML fragment illustrates how the data will be saved in the configuration file and how modules will be started:

```
<SYSTEM>
    <MODULE LOCAL="Yes" USE_FACTORY="No">
        <NAME>LOGGER</NAME>
        <EXECUTABLE>logger.exe</EXECUTABLE>
    </MODULE>
    <MODULE LOCAL="Yes" USE_FACTORY="No">
        <NAME>AUTHENT</NAME>
```

```
            <EXECUTABLE>authent.exe</EXECUTABLE>
        </MODULE>
        <MODULE LOCAL="Yes" USE_FACTORY="No">
            <NAME>REVISION</NAME>
            <EXECUTABLE>revmgr.exe</EXECUTABLE>
        </MODULE>
        <MODULE LOCAL="Yes" USE_FACTORY="Yes">
            <TYPE>GPIB</TYPE>
            <NAME>GPIB_1</NAME>
            <EXECUTABLE>comm.exe</EXECUTABLE>
            <FACTORY_NAME>COMM_FACTORY</NAME>
            <PROCESS_GROUP>1</PROCESS_GROUP>
        </MODULE>
        <MODULE LOCAL="Yes" USE_FACTORY="Yes">
            <TYPE>SERIAL</TYPE>
            <NAME>SERIAL_1</NAME>
            <EXECUTABLE>comm.exe</EXECUTABLE>
            <FACTORY_NAME>COMM_FACTORY</NAME>
            <PROCESS_GROUP>2</PROCESS_GROUP>
        </MODULE>
        <MODULE LOCAL="Yes" USE_FACTORY="Yes">
            <TYPE>TCP</TYPE>
            <NAME>TCP</NAME>
            <EXECUTABLE>comm.exe</EXECUTABLE>
            <FACTORY_NAME>COMM_FACTORY</NAME>
            <PROCESS_GROUP>3</PROCESS_GROUP>
        </MODULE>
        <MODULE LOCAL="Yes" USE_FACTORY="Yes">
            <TYPE>SA3860</TYPE>
            <NAME>SA3860</NAME>
            <EXECUTABLE>acu.exe</EXECUTABLE>
            <FACTORY_NAME>ACU_FACTORY</NAME>
            <PROCESS_GROUP>1</PROCESS_GROUP>
        </MODULE>
        <MODULE LOCAL="Yes" USE_FACTORY="Yes">
            <TYPE>VIRTUAL_ACU</TYPE>
            <NAME>ACU</NAME>
            <EXECUTABLE>acu.exe</EXECUTABLE>
            <FACTORY_NAME>ACU_FACTORY</NAME>
            <PROCESS_GROUP>1</PROCESS_GROUP>
        </MODULE>
    </SYSTEM>
```

The <SYSTEM> section of the file details system wide configuration parameters. Each <MODULE> section details a SCW component as already described in this document.

The module section has two attributes: LOCAL and USE_FACTORY.

- LOCAL determines whether the object exists on the same machine as the Configuration Manager. If it does, the Configuration Manager handles module initialization directly. If it not local, the Configuration Manager passes the startup parameters to a slave server daemon.
- USE_FACTORY allows multiple objects to share the same executable, and passes additional parameters to the executable. If USE_FACTORY=No, the executable is invoked with the object name as the sole parameter. If USE_FACTORY=Yes, the executable is passed additional parameters to determine the actual object incarnated.

Within the module section the additional parameters are as follows:

- NAME is the object name as published through the Name Service.
- EXECUTABLE is the binary to be invoked to incarnate the object.
- FACTORY_NAME is only required for factory creation, and is used to form the object name for the factory published through the Name Service.
- TYPE is only required for factory creation, and indicates the specific object to be incarnated. The type name is exported by the factory, and will be used by the System Configuration Tool.
- PROCESS_GROUP controls the grouping of multiple objects inside the same executable. When several objects share the same object group, the first object causes the factory to be incarnated as well as the object. Subsequent objects in the same process group are created by invoking methods on the factory directly. The process group is combined with the factory name to identify it unambiguously. In the above example each communications adapter is incarnated in a separate process, while the virtual ACU instrument and the physical SA3860 instrument are implemented in the same process.

15.1.2 Slave server

Slave servers exist to offload modules from the main server. This may be for reasons of performance or topology. The following pseudo-code describes how the slave server starts the modules required, based on commands received from the main server. The commands from the main server encapsulate the parameters listed above.

```
(Start SCW service/daemon – absence of configuration file specifies slave server operation)
Register with name server
Loop waiting for module initialization messages from main server. With each message
    If module does not require a factory
        Start module executable with NAME parameter
    Else
        If factory process [PROCESS_GROUP] does not exist
            Start factory process [PROCESS_GROUP]
        Start module via factory process
```

15.2 Communications Adapter Startup

15.2.1 Comm adapter object factory

The comm adapters come in several flavors, and are incarnated through a factory. Some object factories know all about the object they must create, while others use a discovery mechanism. Since the base comm adapter class and its derivations are closely coupled, this executable combines the factory and all the adapter classes in one.

Incarnation of the object is straightforward, therefore, as shown by the following pseudo-code:

```
(Factory started as described above)
Register factory name with name service
Loop waiting for object creation method to be invoked
    Instantiate object of type specified, with name specified
```

The remaining initialization is done within the object itself.

15.2.2 Communications adapter startup

When the object is created, it must configure itself based on configuration parameters obtained from the configuration manager. Then it can register itself and open for business, as this pseudo-code shows:

(Object instantiated as described above)

Locate configuration manager by name service lookup

Request module specific parameters by name

Apply parameters, initialize everything

Verify physical interface, as protocol permits

Register name with name service

Wait for remote method invocations

The specific set of parameters will vary, depending on the protocol implemented by the comm adapter. The following XML fragment would be typical of a GPIB comm adapter, where only the bus number needs to be specified.

```
<GPIB_1>
    <BUS>1</BUS>
</GPIB_1>
```

Other comm adapters could have much more complex parameter sets, such as a serial adapter that would need to specify port, speed and framing.

15.3 Physical instrument startup

Physical instrument objects vary widely in their startup requirements. What is described is generalized high-level behavior that addresses the system interconnections, rather than detailed internal configuration issues. In our sample XML configuration for system startup, described above, one physical instrument object is listed, of type SA3860. This will be described to provide an idea of how other instrument objects will be started.

15.3.1 Physical instrument object factory

The factory operation is a little more complex for physical instruments because of the potential for a much greater selection of derived classes. The factory may know about the virtual instrument (although the mechanism to be described here does not require that) but new physical instruments may be added after the virtual instrument class is developed and the factory executable is built.

An axiom of the SCW architecture is that the binaries should encapsulate single areas of function, and enhancements to other areas of the system should not require a rebuild of all binaries. As support for new instruments is added these can be dropped into the system without affecting existing modules.

Since the physical instruments export a standard personality interface, addition of a new physical instrument would not affect the virtual instrument. The object factory must still know, however, how to map the instrument type saved in the system configuration. The implementation method to support this requirement is that of dynamically loaded DLLs (or shared objects under many Unix variants). Physical instrument implementations will be packaged as DLLs with well-known entries for factory operations. The standard DLL operations would be:

GetPackageDetails which would list the objects supported within the package.

IncarnateObject which is called to create the CORBA object servant for a specific object type.

In addition to the internal identification of supported objects, it makes sense to use a naming convention for the DLL to simplify the selection of which DLL to load to find the object. In our above scenario with the

SA3860 ACU a sensible name might perchance be "SA3860.DLL". Generally speaking, it should be possible to make the DLL name the same as the object type. Where multiple, closely related physical instrument classes, such as variants of a basic instrument type, are implemented in the same DLL an extended naming convention will have to be devised. In the worst case the factory can simply load all DLLs in a set location in turn, and use the *GetPackageDetails()* entry to locate the DLL containing the required class.

Another use for the *GetPackageDetails()* entry is with the system configuration tool. It would not need to built with a knowledge of all the instrument types supported by the system, but it will use the worst-case discovery method described above to extract details of the supported instruments, and offer these to the systems engineer setting up the system.

The actual factory operation pseudo-code is quite straightforward:

(Factory started as described above)

Register factory name with name service

Loop waiting for object creation method to be invoked

  Locate and load physical instrument DLL

  Invoke *IncarnateObject()* method to instantiate object of type specified, with name specified

Once the object is incarnated, all additional initialization is handled by the object.

15.3.2 Physical instrument object

When the physical instrument object is created, it must configure itself based on configuration parameters obtained from the configuration manager. While the instrument specific parameters may be numerous and unique to each instrument, by definition, all will need to attach to a comm adapter. This would be a first step, since the rest of the initialization would almost always include a dialogue with the device itself. Once all the initialization is complete it can register itself and open for business, as this pseudo-code shows:

(Object instantiated as described above)

Locate configuration manager by name service lookup

Request module specific parameters by name

Locate comm adapter by name service lookup

Open virtual circuit to comm adapter, using protocol specific address parameter

Verify device identity

If instrument is not of supported type

  Log fatal configuration error, and terminate

Process remaining parameters, including startup mode

  If startup mode is "reset"

    Apply default and startup configuration parameters

  Else if startup mode is "synchronize"

    Do non-intrusive data synchronization with device

Register name with name service

Wait for remote method invocations

The specific set of parameters will vary, depending on the protocol implemented by the comm adapter. The following XML fragment shows just the parameters used to open the virtual circuit to the device:

```
<SA3860>
   <COMM>GPIB_1<\COMM>
   <ADDRESS>1,19.0</ADDRESS>
</SA3860>
```

Although the protocol specific address is stored with the physical instrument parameters, the object itself makes no interpretation of the address. It is simply a text string to be passed on to the comm adapter, and it will be interpreted by the protocol implementation portion of the sub-classed comm adapter object.

Note also in the pseudo-code the step "*Verify device identity*". Wherever possible, SCW physical instrument objects should interrogate the device once the virtual circuit is established, to ensure that it is of the correct type. The mechanism used to do this will be highly device specific, but many devices support commands to return version information or suchlike that can be used. If this is not available, non-intrusive checking of the parameter settings could give an indication as to whether the device is of the correct type.

15.4 Virtual instrument startup

Virtual instrument objects also vary widely in their startup requirements. As for physical instruments, what is described is generalized high-level behavior that addresses the system interconnections, rather than detailed internal configuration issues. In our sample XML configuration for system startup, described above, one virtual instrument object is listed, of type ACU. This will be described to provide an idea of how other virtual instrument objects will be started.

15.4.1 Virtual instrument object factory

The virtual instrument factory operation is similar to that for physical instruments. It is logical for the factory to know about the virtual instrument although the mechanism to be described here does not require that. A totally generic factory will be developed that always performs discovery by loading and checking the DLLs as described above, but the usual cases will have the virtual instrument classes built into the factory executable.

The following pseudo-code describes the factory operation:

(Factory started as described above)

Register factory name with name service

Loop waiting for object creation method to be invoked

If instrument type is not known to factory

Locate and load virtual instrument DLL

Invoke *IncarnateObject()* method to instantiate object of type specified, with name specified

Once the object is incarnated, all additional initialization is handled by the object.

15.4.2 Virtual instrument object

When the virtual instrument object is created, it must configure itself based on configuration parameters obtained from the configuration manager. Again, the instrument specific parameters may be numerous and unique, but all virtual instruments will need to attach to one or more physical instruments. This would normally be the first step, since the rest of the initialization may be dependent upon the capabilities of the physical instrument. As noted earlier, the virtual instrument layer implements a reductive adaptation layer and adapts itself to the varying capabilities of the underlying physical instrument objects and the devices they embody. Once all the initialization is complete it can register itself and open for business, as this pseudocode shows:

(Object instantiated as described above)
Locate configuration manager by name service lookup
Request module specific parameters by name
For each physical instrument object
    Locate physical instrument object by name service lookup
    Determine instrument capabilities
    If instrument is not of supported type
        Log fatal configuration error, and terminate
Process remaining parameters, including startup mode
    If startup mode is "reset"
        Apply default and startup configuration parameters
    Else if startup mode is "synchronize"
        Do non-intrusive data synchronization with physical instruments
Register name with name service
Wait for remote method invocations

The specific set of parameters will vary, depending on the virtual instrument. The following XML fragment shows just the parameters used to by the virtual ACU instrument to connect to the SA3860 physical instrument:

```
<ACU>
    <PHYSICAL>
        <NAME>SA3860</NAME>
        <TYPE>ACU</TYPE>
    </PHYSICAL>
</ACU>
```

In this case there is only one <PHYSICAL> section in the configuration. Other virtual instruments may contain multiple <PHYSICAL> sections if their functionality is implemented across a number of physical instrument objects.

The <TYPE> parameter is used by the virtual instrument to narrow the CORBA reference to the physical instrument tot the correct type. Normally, physical instrument objects of the same type will inherit from a standard mix-in class and export a common interface. Variations will be resolved within the physical instrument object. Where the variations are more profound the adaptation will need to take place in the virtual instrument. This is especially the case where two or more physical instruments are combined to provide the same capability set that may be implemented in a single device of another type. The virtual instrument object will deduce this situation from the type of the physical instrument. If the <TYPE> parameter specifies a reduce capability device, the virtual instrument will look for another <PHYSICAL> section identifying an instrument that provides the missing functionality.

16 Scheduler Virtual Function

The Scheduler Virtual Function is responsible for managing timed, unattended events. It provides a general purpose scheduling function, and makes no assumption about the nature of the tasks being scheduled. All characteristics of the task are expressed through parameters and attributes contained in the XML schedule file. Tasks are atomic, and once commenced run to completion unless explicitly aborted.

In addition to initiating operations at the appropriate time it is also responsible for monitoring resource allocation. Tasks are prioritized, and higher priority overlapping tasks requiring the same resources will prevent lower priority tasks from being initiated. A flexible prioritization scheme allows tasks to be prioritized by time of scheduling, source of task request, and explicit priority assignment.

Multiple tasks may be executed simultaneously if they do not conflict for resources. Schedule updates may be made affecting any running tasks, unless a new task is scheduled at a higher priority and requires the same resources as a running task. In this case the running task may be preempted, if preemption is enabled for the system.

16.1 Scheduler Objects

Two types of objects are maintained: task objects that describe unattended operations, and resource objects that maintain a timeline for each resource in the system.

16.1.1 Task Objects

Task objects implement a three tier hierarchy of timed objects:

- A **Job** is the highest level aggregation of operations. This would encapsulate all that needs to be done during a satellite pass in a LEO system, for example. The job level is where the resources are listed, and must be successfully allocated before the job can commence. Jobs would typically fall into one of three categories: *work*, which would mean handling a satellite pass in the current application, *housekeeping*, such as archiving data to tape, data collection services or software updates, and *maintenance*, such as running hardware tests, or scheduling down-time for unit replacement.
- An **Operation** is complete task to be performed, such as a data download or upload. A **Job** may be comprised of many operations, and the failure of some may not affect the rest. An **Operation**, on the other hand, is an “all-or-nothing” affair – it either succeeds or fails. Some operations may be categorized as “critical”, and if they fail the job should abort. Other operations can be made dependant on the completion of other operations, to ensure sequential operation.
- An **Event** is a single step in the performance of an **Operation**.

The scheduler is concerned primarily with jobs and operations. Events are handled within other virtual functions that are incarnated for the duration of the operation, and which translate events into specific commands for virtual instruments.

16.1.2 Resource Objects

Resource objects monitor module use by compiling a list of start and stop time pairs for each object by name, and sorted by start time. It is not necessary to know anything about the resource at this level. If the start time of one element precedes the stop time of the previous element, a resource conflict exists.

In some complex installations there may be multiple similar or identical devices allowing simultaneous operations. These are selected by programmable switches. To support flexible management of these instruments, there are also ResourceSet objects that group like instruments into resource pools. A resource set will list *inputs*, *outputs*, *selectors* and *instruments*. Instruments will be virtual, and may well be aggregations of multiple devices. Selectors will also be virtual instruments, and will implement N→M switch banks.

16.2 Schedule Creation

In its simplest case, schedule creation is a matter of listing times, devices and device operations.

The present remote sensing software treats the station as a single entity, and assigns all the resources in the system to the job. The only issue is whether all the devices are online. The scheduler manages four stages of a pass: prepass test, prepass, track, and postpass. Instrument settings are handled by the use of

configuration files that contain settings for all the instruments. This are applied at prepass time. In addition, a separate scheduler handles recorder events.

The scheduler implemented for the example system will conform to this behavior, and combine the role of pass scheduler and recorder scheduler. A typical set of operations supported would be:

- Prepass test
- Prepass – this is a critical operation
- Track – this overlaps uplink and recorder operations
- Uplink
- Recorder operations
- Postpass

Events would correspond to the application of a configuration to the devices, as well as start and stop commands for various functions of the virtual devices.

The main scheduler thread is the sole point of access for the configuration file. All jobs to be added to the schedule must be prepared elsewhere, either by a user with a GUI, or by some ingestor module (a virtual function) that receives scheduling data from a remote system and formats it to suit the internal structure of the schedule. The job definitions will be sent to the scheduler as CORBA structures defined in IDL, and would consist of lists of operations with embedded lists of events, and a list of resources required for the job. The scheduler translates the CORBA structures into XML and saves it to the file. If the new job is due within the scheduler's cache window the job would be added to the memory cache of jobs.

Periodically the main scheduler thread scans the file and updates its job cache, and removes old schedule data. If jobs are implemented as recurring, the scheduler inserts the next instance of the job into the schedule. Recurring jobs can only be specified by time. Automatic calculation of passes based on ephemeris projection would be handled in a separate virtual function, and the calculated job passed to the scheduler as described above.

Individual instrument settings will still be handled by configuration files. These are relatively static and may be used many times for repeated operations, such as satellite passes, system backup, and the like. Configurations are created through the user interface and managed by the configuration manager.

16.3 Schedule Playback

16.3.1 Main scheduler thread

The scheduler will be monitoring real time, and the schedule file, to determine when a job should begin. When a job is due to start it reserves the resources required, and creates a separate job thread object which is responsible for executing the scheduled operations.

Should the scheduler or the job thread object reserve the resources? If we want more granularity of allocation in the future it would be better to defer this to the job. Also, if the scheduler acquires control of the resources, and that control is subsequently revoked, the scheduler would have to call into the job the abort it. On the other hand, if a higher priority job overlaps the running job the scheduler may well require such a mechanism.

The only other communication between the scheduler and the job thread object is upon termination or completion of the job.

16.3.2 Job thread object

The job thread is driven by a real-time ticker event and initiates operations as they become due.

17 Node Controller Virtual Function

18 Design Philosophy

The underlying philosophy of the design is get away from the complex interrelationships between modules that make it so hard to expand and add features to the current system. Using the facilities provided by CORBA it is possible to create a highly compartmentalized system, with many modules, each doing one thing well. By defining the interconnection points as standard interfaces, the inner workings of each module are isolated from the system, and may be enhanced or modified at will. The key to the success of the architecture is the proper definition of the interfaces, making them rich enough to be stable over a long period, while keeping them as simple and easy to implement as possible.

Another feature of the architecture that improves the compartmentalizing of function is that all objects are self-referencing and self-configuring. In the case of a physical instrument, for example, the ideal time to define and code the parameter types and ranges is when the instrument support is written, and the best place to store it is right in the instrument class. This allows a new instrument to be added to the system without affecting other modules. Once the new instrument implements a standard interface, it will “plug-and-play”.

Using class inheritance, it is easy to add functionality to many instrument types. An ongoing consideration in all design reviews will be determining whether added functionality belongs in the derived classes or the base classes. Where the addition of one or two parameters will provide a generic function instead of a specific one the choice should always be to opt for generic behavior. These generic functions can be enriched later, using the additional parameters to preserve backward compatibility.

These philosophies give birth to a number of paradigms that must be observed in the implementation.

18.1 Paradigms

- Generic functionality belongs in the base classes.
- An object must know everything about itself and as little as possible about everything else.
- No magic numbers, no literal numbers
- No referencing of instruments by index across module boundaries
- Never assume buffers are large enough. All copy/append operations must perform bounds checking.

[More TBD]

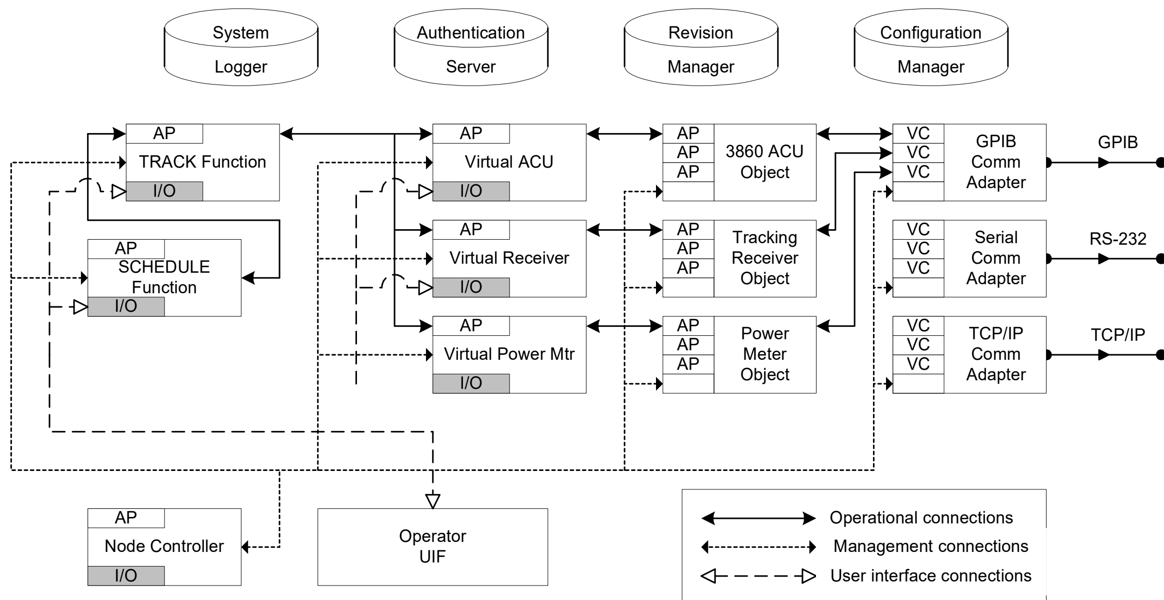
19 Example Implementation

To illustrate how the various modules interact, the following example implementation is proposed: a simple system with a 3860 ACU and a tracking receiver, performing LEO tracking and data reception. The devices listed are all GPIB attached, so the modules required would be:

- ☑ Authentication Server
- ☑ System Logger
- ☑ Configuration Manager
- ☑ Revision Manager
- ☑ Operator UIF
- ☑ GPIB Communications Adapter
- ☑ 3860 ACU Physical Instrument Object
- ☑ Tracking Receiver Physical Instrument Object

- ☑ Power Meter Physical Instrument Object
- ☑ ACU Virtual Instrument Object
- ☑ Receiver Virtual Instrument Object
- ☑ Power Meter Virtual Instrument Object
- ☑ TRACK Virtual Function Object
- ☑ SCHEDULE Virtual Function Object
- ☑ Orbital Propagator Virtual Function Object
- ☑ G/T Test Virtual Function Object
- ☑ Node Controller Virtual Function Object

A block diagram representing this implementation is as follows:



To change the physical devices to different types would merely involve replacing the physical instrument objects. If the classes were already written then no coding at all would be required, and the changes could be made on site by a technician. If the new physical instrument class did not exist only the device specific code would need to be written; the generic behavior and instrument personality would already exist in the base classes. This exercise could be as simple as coding new command strings.

20 SCW Design Requirements

20.1 Design Drivers

This section lists the prime . drivers for the design requirements, and reviews how the SCW architecture satisfies them.

1. **OS independent, only Windows XP implemented initially. Reporting and hardware communication may not be OS independent.**

[TBD]

2. **The system will be able to be controlled or observed from multiple locations simultaneously. The various locations will be able to control and observe the system from multiple levels, as both an aggregate system or as a collection of instruments. Multiple, simultaneous observers will be allowed, however only a single controller will be allowed per functional block.**

   Polled inquiry of object parameters is simple, once the reference to the object can be resolved. Thus any UIF or control agent that knows or can discover the name of an object, and can satisfy the authentication requirements can monitor and control devices within the system. The issue of preventing multiple control will be handled within the authentication service. The CORBA Event Service provides a publish/subscribe mechanism that will allow multiple locations to received push data from any object with that capability. The node controller virtual function supports aggregation of instruments.

3. **A low level of expertise should be required to operate, upgrade, install, and configure the system. An operator with appropriate permissions and a user manual should be able to perform standard system tasks. Software engineering support should only be required for exceptional cases or new functionality.**

   This requirement actually addresses three areas:

   Ease of *operation* is largely a function of an intuitive UIF. The profile-driven configurable UIF engine makes it easy to define simple operator interfaces for cases where ease-of-use is required. The same engine with a different profile can activate a sophisticated administrator interface.

   Ease of *installation and upgrade* is provided by the modular nature of the system, and the self-referential capabilities of the objects. Installation scripts or utilities will use this information to copy the required modules to the system. This will ensure that all modules are compatible.

   Ease of *configuration* is provided by the System Configuration Tool. As this evolves into a fully-fledged drag/drop utility, system configuration will closely match the block diagram of instruments and connections.

4. **User should be able to configure the state of and the GUI representation of system instrumentation. The user should be able to store the configuration they have created.**

   This is provided by the user interface GUI engine.

5. **User access control (different access levels and Discretionary Access Control (DAC) to data).**

   This is provided by the authentication server.

   .

6. **Remote monitoring and diagnostics.**

   This is provided by the CORBA infrastructure, and the exported parameters.

7. **Multiple platform support**

   This is provided by the inherent cross platform features of CORBA.

8. **The system will use industry standard interfaces and protocols whenever possible for communication between system components.**

   The principal protocol used in the system is CORBA, which in turn uses TCP/IP. Remote control protocols will be determined by system demand, but the SNMP and HTTP protocols are planned.

9. **Timing accuracy of better than 250 milliseconds will be provided by hardware.**

   Where the functionality is distributed over several computers, all will have to use some time synchronization protocol such as NTP or GPS. Modules will then use the system time functions for timing purposes.

10. **On demand and scheduled track events.**

    By implementing TRACK as a separate virtual function, and allowing multiple control of virtual functions, the TRACK function is accessible to the UIF and to the scheduled operation. Depending on the

authentication level of the UIF either control source could take precedence. Typically, an operator would have a lower priority than the scheduler would, while an administrator would be able to override the scheduled operation.

11. **Local and remote capability for all system features.**

    With CORBA, all interfaces are accessible remotely.

12. **Scalable from single instrument to multiple terminals.**

    This is provided by the modular architecture in general and the node controller specifically.

13. **Extensive logging and data recording capabilities.**

    This is provided by the built in logging features of the base classes and the System Logger.

14. **Proper allocation of functionality between ACU and station software**

    [TBD]

15. **Support 3840 and 3860 ACU's**

    [TBD]

20.2 Operational Functions

20.2.1 LEO/MEO Driven Functions

Scheduling

- Contact Scheduling
- Event Scheduling
- Scheduling multiple SV's autonomous passes
- Scheduling multiple SV's with priority
- Equipment Configuration

Tracking

- Acquisition
  - None
- Track
  - None
- Reacquisition
  - None

Logging

- Pass data log. Configurable on or off on a per pass basis.

Equipment Control

- Equipment synchronization during a pass (pseudo real-time coordination)

20.2.2 GEO Driven Functions

Scheduling

- Single Satellite tracking unless turned off
- Random arrival of new ephemeris

Tracking

- Acquisition
  - None
- Track
  - Continuous Track (using any mechanism)
  - Long-term operation
- Reacquisition
  - None

Logging

- Continuous data collection
- Configurable continuous data collection

Operations

- Customer unique link setup

20.2.3 Common Functions

Scheduling

- Script Scheduling (Tests)

Tracking

- All features configurable per satellite (Acquisition method, Track method, and Reacquisition method)
- Acquisition
  - Dwell at acquisition point
  - Scans
  - Time shifting (Manual or Commanded)
- Track
  - Autotrack Mask
  - Orbit propagation in SW or ACU
  - Step Track
  - Autotrack
- Program Track
  - Ephemeris
  - File of pointing angles
  - Data Stream
- Reacquisition
  - Scan
  - Step
  - Time Shift

Logging

- Status messages
- Error messages
- Test results

- Equipment messages
- Configurable data gathering (ideally configurable per pass or function). User will be able to configure what data is collected on a per functional group basis. Configuration will be storable.
- SW trace log
- Master event log
- Integration of operating system logs
- User actions
- Configuration changes
- Security audit events

Automated Testing

- Servo test suite
  - Small Step
  - Large Step
  - Ka
  - Kv
- G/T tests
- BER tests (Single point or Curve)
- Optical Alignment
- Alignment Algorithm
  - Boresight test suite
  - Jitter Test
  - Error Gradient Test
  - Programmable pull-in test
- Positioning Fault Isolation Suite
- Fault Isolation Suite for symmetric systems (multiple loop back paths)
- Fault Isolation through redundancy

Equipment Control

- Manage redundant units when present
- Polling, interrupt or combination methods for instrument status, alarms and data
- Integrate CFE alarms and signals
- Constant Polling or alarm message recognition from instruments
- Redundancy management

20.3 Management Functions

20.3.1 System Operations

- Autonomous Operation or on demand
- Remote shut down/startup

- Maintain and control system status, mode, state

20.3.2 Fault Management

- Automatic fault detection
- Fault correlation
- Recognize fault clearance from unit
- Recognize fault clearance from unit replacement
- Automatic and on demand fault isolation suites
- Suppress fault reporting on demand

20.3.3 Configuration Management

- Recognize instruments as they are removed/replaced
- Inventories of HW and SW
- Recognize HW FW versions
- Recognize units removed and replaced
- Selection of SW version
- Multiple SW/FW Version maintenance
- Local and remote SW update
- Devices need to be able to be taken on/off line without affecting unrelated functions.

20.3.4 Performance Management

- Constant link data collection
- Analysis of data against thresholds
- Data analysis tools

20.3.5 Security Management

- Control user scope with levels of access
- Protect files with discretionary access control (TBD)
- Arbitrate local/remote control
- User identification is managed in conjunction with OS security.
- User interface will allow access by function based on user access privileges.

20.4 User Interface

- Corporate logo should be prominently displayed.
- Basic GUI display with no log in
- User will have a choice of a summary view, block diagram view, or a cockpit view as the top-level screen.
- Instrument screens will have separate sections for controls, status, and data. Each will be individually resizable.

- All software functions will be accessible through the menu bar directly.
- GUI components will be co-resident on the same machine with associated back-end functionality.
- System functions will each have their own, separate screen.
- GUI will allow the viewing and exporting of data files.
- Administrator will be able to administer data from the GUI.
- User will be able to configure data display and layout in later versions of SCW.
- User will be able to display real time data as a numeric data field, bar graph, strip chart, or spherical view.

20.5 General Design Requirements

20.5.1 Scalability

- Must work from minimum case (computer and single instrument) to the maximum case (groups of antennas with NASA style systems).
- Provide good performance from the minimum case (computer and single instrument) to the maximum case (groups of antennas with NASA style systems).

20.5.2 User SW Maintenance

- Auto-start on boot will be user selectable.
- Auto-elimination of software log files based on creation date will be user configurable.
- Auto-elimination of software log files based on disk space usage will not be user configurable.
- A utility for switching what revision of software is being used will be provided.
- Logged data will be divided into two categories; one will be data that is always logged that the user cannot disable, the other will be data that the user can determine whether it is logged or not. Data that is always logged will be determined by what is viewed as necessary by Scientific-Atlanta for customer support.

20.5.3 Distributed Architecture

- Software should work on a single machine or across multiple machines.
- Software should not require any communication connectivity other than TCP/IP network stack.

20.5.4 Installation and Upgrade Processes

- Simple Installation/Upgrade process. A non-technical person should be able to perform the process without supervision.
- Software should be able to determine that all of its components are from the same release and provide a warning to the user if this is not the case.
- Upgrade process should schedulable (i.e. system should be able to upgrade itself during a time when passes are not scheduled).
- Upgrade process should be able to be performed remotely.
- Software must be restarted for the new software revision to be used.
- A machine reboot will not be required for installation of new software.

- OS patches and OS driver updates are not considered part of a software upgrade.
- OS files should be generated by users using SA directions and OS tools.
- SA will be able to work in any network configuration.

20.5.5 Remote Protocols and Interfaces

Scheduling

- Customer provided file that supports appending to, replacement of, and cancellation of the schedule.

Ephemeris

- Update from customer computer
- Update via Internet
- Update via Scientific-Atlanta dial-up service

Control/Data Reporting

- Status Packets (NASA)
- Equipment monitoring
- Equipment control
- SNMP
- Data provided will be either real-time or a file

20.5.6 Data Management

- Configuration data will be viewable natively in an XML viewer.
- System will be able to maintain at least 30 days of log files.
- Data will be stored in a format that is understandable with commonly available utilities (for example, Excel).
- Data will be stored using the standard ASCII character set.
- Data will be directly viewable in MS Excel; standard macros may be developed to assist in data comprehension.

20.6 Design Strategy

20.6.1 Object Oriented

- Use C++/Java for increased modularity.
- Functionality will be localized to a single module; i.e. all scheduling functionality will be in a single module.

20.6.2 Adaptable

- If an object is used for its previous function, (i.e. no new requirements are added) no modifications should be required to the object.
- Objects should be usable even after modification in previous versions without modification to either the new object or previous versions.
- Objects, such as instruments, should be derived from standard templates.

20.6.3 Small Pieces

- Modules should be the smallest self-consistent functional block possible i.e. a scheduler instead of a single, monolithic ground station module.

20.6.4 Divorce User Interface from Instrumentation

- Changing the vendor for a particular instrument should not require GUI modifications. GUI modifications should only be required for new features or functionality.
- Assuming the parameter is already present in the instrumentation, displaying a new instrument parameter should not require changes to the instrumentation code.