Reliable Control System For Future Particle Accelerators

DRAFT 14/09/2009

Artem Kazakov

Department of Accelerator Science
School of High Energy Accelerator Science
The Graduate University for Advanced Studies

2009

Table of Contents

Introduction

Modern particle accelerator machines are complex and large scale structures. Large projects like Large Hadron Collider and International Linear Collider (ILC) consist of thousands of components that are spread over big distances in underground tunnels[1]. Machines of that scale and complexity raise a set of challenges for all subsystems of the accelerator. With constantly growing size and complexity of particle accelerators the role of the control system becomes more and more important for a successful operation. One of the biggest concerns for large machines is availability. Because of a huge number of components, even very reliable components, final availability of the accelerator might suffer of continuos failures in one of the subsystems.

Availability	*Downtime Per Year*
90%	36.5 days
99%	3.7 days
99.9%	9 hours
99.99%	53 min
99.999%	5 min
99.9999%	32 sec

Table 1. Availability and downtime per year

For example target availability for the International Linear Collider is 75%[1], but in order to achieve that, the control system has to be available for 99-99.9% of time (15 hours of down time is “allocated” for the control system of the ILC. See Table 1). Design draft specifies that the ILC control system will consist from ~1200 “crates”, and that translates into 99.999% availability for each crate. Such availability has not been a requirement for present accelerator control systems. Therefore it sets a new challenge for control system designers, implementors and operators. A multilevel systematic approach should be taken in order to achieve these availability goals.

Introduction

Lets analyze availability indicators for current accelerators. Typical high energy physics accelerator currently has an availability of 75-85%[2]. Though there are some examples of much better availability: Pohang Light Source (2008) - 97% (with controls responsible for 2% of downtime)[3]; SOLEIL light source (2007) - 95.7% (with controls responsible for 2.7% of downtime)[4]; KEK Linac (2008) - 98.3% (with controls responsible for 13.3% of downtime)[5]. For KEK Linac it means that control system availability was around 99.76%. The KEK Linac control system consists of 30 VME crates, 150 PLCs, 30VXI, 15 CAMAC, 24 intelligent oscilloscopes[6]. ILC control system is 1195 ATCA crates, 8356 network switches and thousands of other lower level control components. ILC control system has 10 ~ 100 times more components than KEK Linac control system. Such tremendous increase in a number of components will dramatically reduce the availability indicators for the control system. Therefore availability issues have to be seriously considered for the future particle accelerator control systems. This research was devoted to that particular goal.

The first chapter provides some introductory information regarding accelerator control systems, historical overview of the control system evolution and a modern view on building control systems.

The second chapter of this work describes different approaches to improve availability of a particle accelerator. The general reliability theory is briefly introduced. Then the applications of that theory to accelerator control system are discussed. An accelerator control system can be roughly separated into four major parts: hardware, software, humans and procedures. Analysis is done for each of these four components. Each of these parts requires different approaches in order to achieve high availability. This work covers improvement of the software and hardware components. Hardware reliability is improved through implementation of redundancy, and software reliability is improved through implementation of a test system, that ensures the software quality. Further chapters provide more detailed explanation on how these goals are met.

Chapters 3,4,5 describe my contribution to improve reliability of accelerator control system. This work is mostly concentrated on improvement of software and hardware components using EPICS software. EPICS stands for Experimental Physics and Industrial Control System. It has more than 15 years history of usage and has been being developed during all these years. It is widely used in many accelerator laboratories all over the world, including KEK, where it is a basis for the KEKB control system[7].

Chapter 3 describes the EPICS redundant IOC. In order to achieve high availability (such as 99.999%), redundancy is essential (as discussed in Chapter 3, section 3.1, 3.2). Redundancy allows to reduce time needed to recover from a failure to a few seconds or milliseconds, instead of hours and days. The system is not stopped because of the failure and the stand-by component starts to operate immediately after the failure is noticed. The redundancy approach is a common technique used in highly available applications (more than 99.999% availability)[8]. The original EPICS software distribution lacked redundancy support. This issue was addressed by developing EPICS redundant IOC. The initial design and development was made by DESY[9]. Unfortunately from the very beginning only vxWorks support was looked for. Later it was realized that other OS support is needed as well. As a part of my research, in collaboration with DESY, I generalized the redundant EPICS IOC to Linux, Darwin, and other operating systems[10]. The generalization was done using the Operating System Independent (OSI) library, therefore the ported version should work on any platform, where the OSI library is fully implemented. The generalized redundant IOC is an important improvement to the EPICS control system framework. Several serious software bugs were fixed in the original Redundancy software. The result of this work is a very important improvement to the existing RIOC implementation. First, it allowed to use RIOC on many operating systems, such as Linux and MAC OS X, therefore providing much wider application field for the RIOC. Second, it allowed to include the support for the RIOC into the official EPICS distribution from version 3.14.10[11]. Third, working on this project resulted in modification and splitting the original software into several libraries which can be used independently. An example of such usage is provided in the next chapter, describing the implementation of redundant and load-balancing Channel Access gateways. In chapter 6 the generalized version of the RIOC is extended to support the Advanced Telecom Computing Architecture (ATCA) platform. These projects would have been impossible without the generalization of the original RIOC and improvements done during this work. To summarize: the EPICS redundant IOC was designed and developed by DESY[9], but it was only available for vxWorks operating system. I generalized the EPICS redundant IOC to other operating systems and made it available for wider range of applications[10]. Within this work the redundant EPICS IOC was implemented on Linux, Mac OS X and Solaris systems for the first time.

As mentioned above using the generalized versions of RIOC libraries, Channel Access Gateway was made redundant and load-balancing. This new and original development is described in Chapter 4. Channel Access gateways are in operation in many places. They allow separating control networks into several administrative subnetworks. Also they can be used as a security tool: providing restricted access to the control network, for example read-only access from public networks. Besides this administrative and security aspects gateways also optimize the number of Channel Access (CA) connections to the IOCs, because several CA clients can share one connection to an individual IOC. Due to these important functionalities gateways play a growing role in today's installations. Performance and functionality have been continuously improved over the last years. The availability of this service is key for machine operations in many places. This was the driving force to implement redundancy also for the CA gateways[12]. The Channel Access gateway originated at APS by during the years has been developed by Jim Kowalkowski. Further development was done by several people at APS, LANL and BESSY[13]. I implemented the redundant and load-balancing Channel Access gateway, based on the original Channel Access Gateway.

The redundant and load-balancing Channel Access Gateways were implemented within this research. The development was done using the generalized version of RIOC libraries discussed in the chapter 3. The implementation of the redundant CA gateway allowed to escape the single-point of failure, and by introducing the load-balancing the performance and throughput was improved in the number of 2. Load-balancing version of the CA gateway brings availability improvements as well, due to the fact that half of the connections are handled via the secondary gateway, these connections will not be affected when the failure occurs on the primary gateway.

In chapter 5 new hardware standard ATCA and its application to control system are described. Within this research a support for Advanced Telecommunication Computing Architecture (ATCA) was added to the RIOC software. This addition, called ATCA-driver, allows to monitor the ATCA-hardware and provide better availability. This driver can help predict hardware failures and allows to decrease the Channel Access clients reconnect time from 30 to 2 seconds.

Introduction

Using reliable software in conjunction with reliable hardware can give us even more reliable solutions. Recently Advanced Telecommunication Computing Architecture standard is gaining attention in High Energy Accelerator field as platform for modern controls and Data Acquisition (DAQ) systems. ATCA is an open standard developed by consortium of telecom vendors and users; and from its very early days it is aimed to high reliability, high bandwidth and modularity. Nowadays it is widely used in telecom industry and is widely supported by many big vendors. The ILC control system requirement for a single control shelf is 99.999%. ATCA hardware available on the market provides this level of availability or better, therefore ATCA was suggested to be used in ILC control system. Even though ATCA provides redundant cpu boards, power supplies, interconnections and other facilities, redundant IO boards and software that can work with ATCA hardware and utilize its capabilities must be developed[1].

For the reasons mentioned above I developed a driver for the Redundant EPICS IOC (RIOC) which provides support for ATCA. Using the Hardware Platform Independent library (HPI) it allows the RIOC to monitor the status of the hardware it's running on. Using this information, fail-over decisions can be made even before the "real" failure happens. For example, if the temperature starts to rise there is some delay until system crashes because of overheating. During that time the fail-over sequence can be triggered. Therefore the fail-over happens in a more stable and controlled environment. An obvious and very important benefit is that client connections can be gracefully closed and clients would reconnect to the stand-by IOC within 2 seconds. In case of a real hardware failure it would take up to 30 seconds (default Channel Access timeout).

This ATCA RIOC driver can be also utilized on any computer which has an HPI support. Therefore providing increased availability for the platforms other than ATCA. For example modern computer severs are usually equipped with the temperature, voltage etc. monitoring hardware. OpenHPI distribution of HPI supports such hardware on Linux operating system[14]. The ATCA and HPI library are designed and created by other people. The ATCA RIOC driver was developed by me.

Chapter 6 presents another approach to improve the reliability - by means of improving the quality of the software. An important part of this process is the software testing and quality assurance. In early years EPICS supported only one operating system for the server side - vxWorks, and one operating system for the client side - Sun OS; and it was well tested at

Argonne National Laboratory. But in recent years EPICS has gained support for many operating systems and hardware platforms and now it supports more than 10 operating systems[15]. Each institution uses its own collection of OS+EPICS running on different hardware. Most of these combinations are not very well tested, due to a lack of convenient, easy to use and reliable system integration testing mechanism. Therefore it leads to a potentially dangerous situation when an untested and unproved software is used for the operation. Obviously a decent automated test system is needed for EPICS software distribution[16].

EPICS has a decent unit-test system included in the base distribution. It has been continuously extended by core-developers as EPICS evolve and new features have been added. Basically a unit-testing is a testing of small pieces (functions) of code, to check that they perform correctly. But it does not mean that these pieces would work when combined together. For that purpose there is another test package, which consists of system tests, when real IOC's are installed on distributed machines and then it is checked if these systems perform correctly altogether. The unit tests and the system test software was developed by EPICS community. I designed and developed the original software for automating the system test procedure.

Originally that system test package for EPICS consisted of several programs/IOCs and text file instructions how to run them. But it is not convenient and takes a lot of time for developers and users to understand how to run these tests, prepare different machines, upload, configure and start the IOCs, preform a test and compare the results. As a part of my research I have developed a system that automates the process of a system testing and system integration testing, and provides a flexible environment to create these tests. This newly developed software supports a wide variety of configurations by default and can be easily configured using simple configuration file. It is developed in the high-level object oriented scripting language Ruby, which makes it easy to extend and add new functionality. Usage of this automated test systems greatly simplifies the testing process. It allows to run a predefined set of tests on a predefined set of computers in a fully automated manner. It requires to create a configuration file, specifying the computers and corresponding test that must be performed. Then only one command must be issued by human to run all the tests. If compared to manual testing, it saves tremendous amount of human time and effort. Due to automation chances for human error are greatly reduced too. After all tests are executed the system provides a detailed report.

1 Control Systems overview

Nowadays, a control system of a particle accelerator is a very important subsystem playing a significant role in operation of the machine. But it was not always like that. First particle accelerators did not have any computerized control system and were operated using analogue devices and interfaces. When the computer industry developed enough, relatively cheap and powerful computers became available in the late sixties. As computers became affordable for physics laboratories, physicists started to experiment with the new possibilities that computers provided. This was the beginning of the computerized control system era for particle accelerators. Since its conception a computerized control system evolved through the following periods:

- non-centralized computer adjuncts to the accelerator
- centralized computer control, vendor-oriented, with very substantial custom solutions (example PDP-10, shown on Figure 1.1)
- distributed networked computer control, standards oriented,with mostly commercial solutions

Figure 1.1: PDP-10 the centralized control system.

These eras are not, however, mutually exclusive; even today systems characteristic of the earliest period can still be found, even at the newest accelerators. Of course, interwoven through these periods is the history of the computer industry itself, from mainframe computers to mini-computers, to workstations, to personal computers, to commodity single board computers. Today, accelerator operators and physicists assume that a powerful computer control system should be part of an accelerator. The success of a controls group in delivering high capability controls solutions that are transparent and user friendly is a major component of what a controls system can contribute to the successful operation of a facility. Another important role of an accelerator control system is to provide a tool which allows the scientists to implement their ideas and theories in hardware. For that purpose an accelerator control system must provide flexible tuning capabilities, different abstraction layers and simulation capacity[17].

Most modern particle accelerators control systems are distributed computer systems, built on open standards. Current trend is to use more “commercially of the shelf” available components and solutions. The distributed control system provides a great flexibility, computational power and scalability; on the other hand it is more complex than the centralized model and raises a lot of challenges for the designers, implementors and maintainers. Some of those problems rise from the distributed nature of the system. Along with a greater scalability the problem of the reliability rises. Even if the most reliable components are used, when we combine thousands of these components the probability of a failure rises drastically. Another issue emerges from the physical distribution over a large space, often not easily accessible for the service and maintenance, and those factors play an important role in the system reliability. Therefore distributed systems require a careful design if the high reliability is sought. Overall reliability constrains and maintenance requirements for an accelerator control system are more strict than for the other accelerator subsystems. Normally during the year there are one or more maintenance periods during which most of the subsystems are stopped. Nevertheless the control system is often needed to be operational even during these maintenance periods in order to do the maintenance of the other subsystems. Therefore when the control system is designed these considerations should be taken into account.

1.1 The "standard model" of accelerator control system

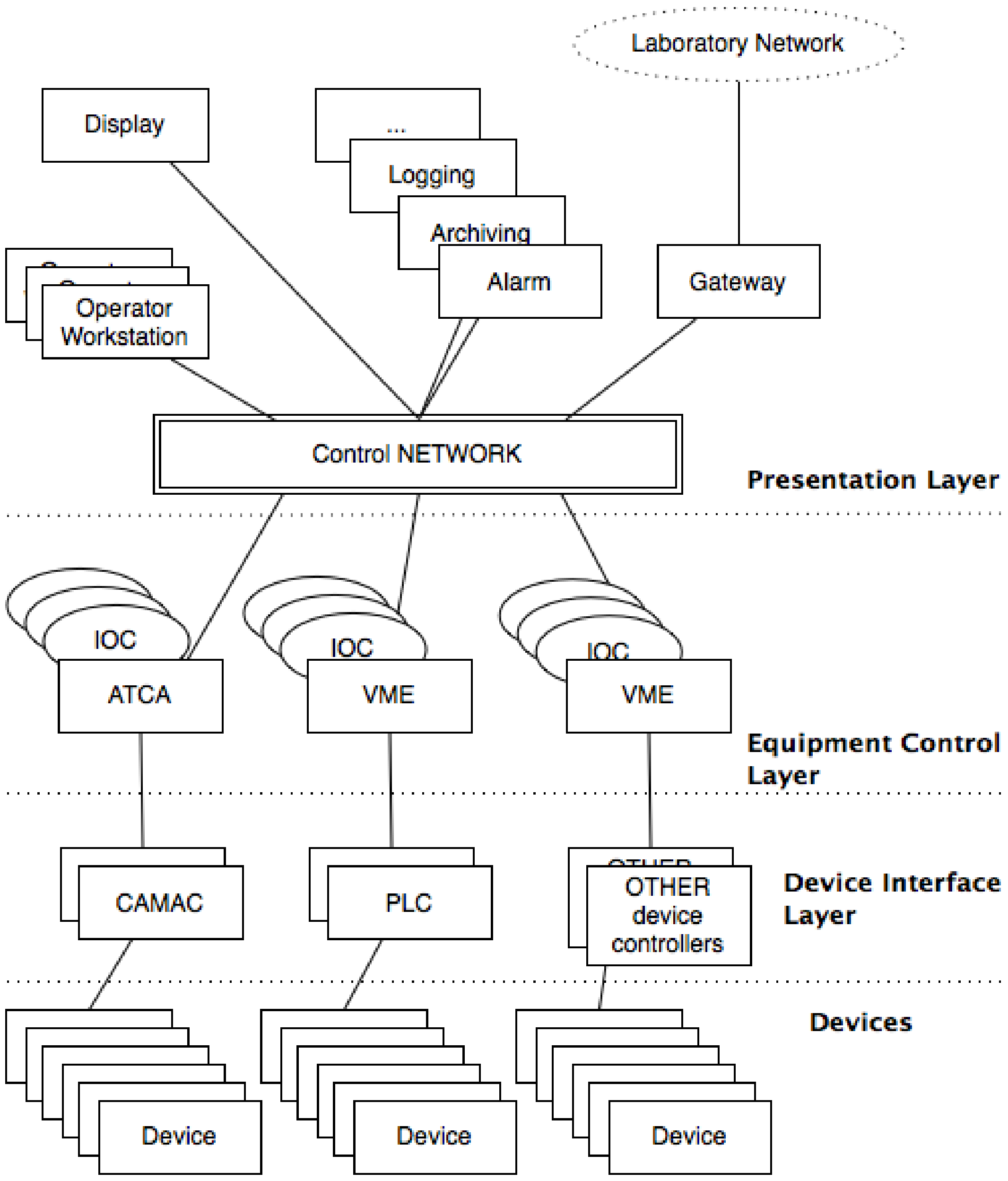

Figure 1.2: The standard model of a control system.

Most modern distributed accelerator control systems incorporate so called "Standard model" for the accelerator control system (Figure 1.2). The standard model defines three layers of abstraction of the controlled equipment:

- Presentation Layer
- Equipment control Layer
- Device Interface Layer

The highest level of abstraction is the Presentation Layer, includes operator consoles, simulation systems, archives, log managers, alarm handling and other components. The operations software runs at this level. An example of Presentation Layer can be seen on Figure 1.3.

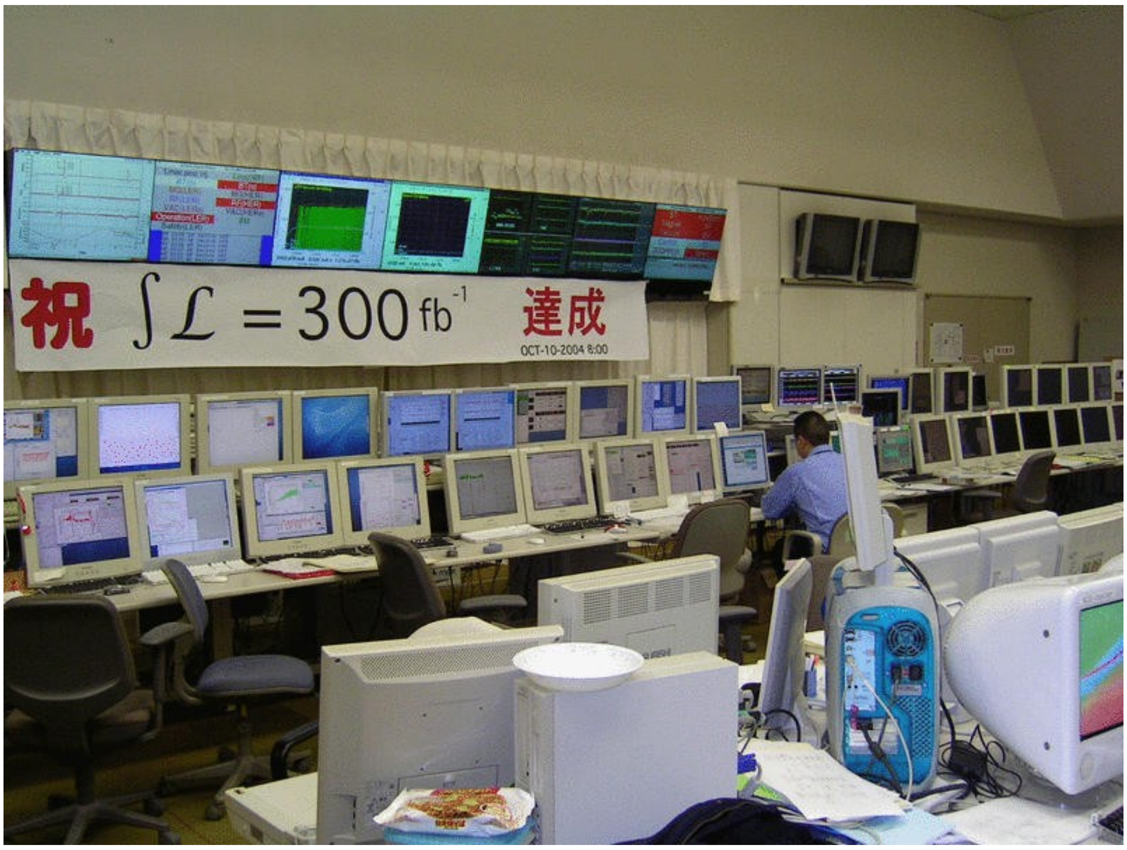

Figure 1.3: Presentation layer - operator consoles in KEKB control room

Figure 1.4: MVME5500 cpu board

The Equipment control Layer consists of Input Output Controllers (IOC), that are connected to the device controllers via different fields buses, which belong to device interface layer. Typically these IOC controllers are VME cpu boards (Figure 1.4), running real-time operating system like vxWorks, or general purpose operating system like Linux. An example of such system representing the Equipment Control Layer is shown on Figure 1.5. It worth noting that modern developments in embedded technology allows to run IOC and device controller on the same chip. That makes Device Interface Layer and Equipment Control Layers indistinguishable.

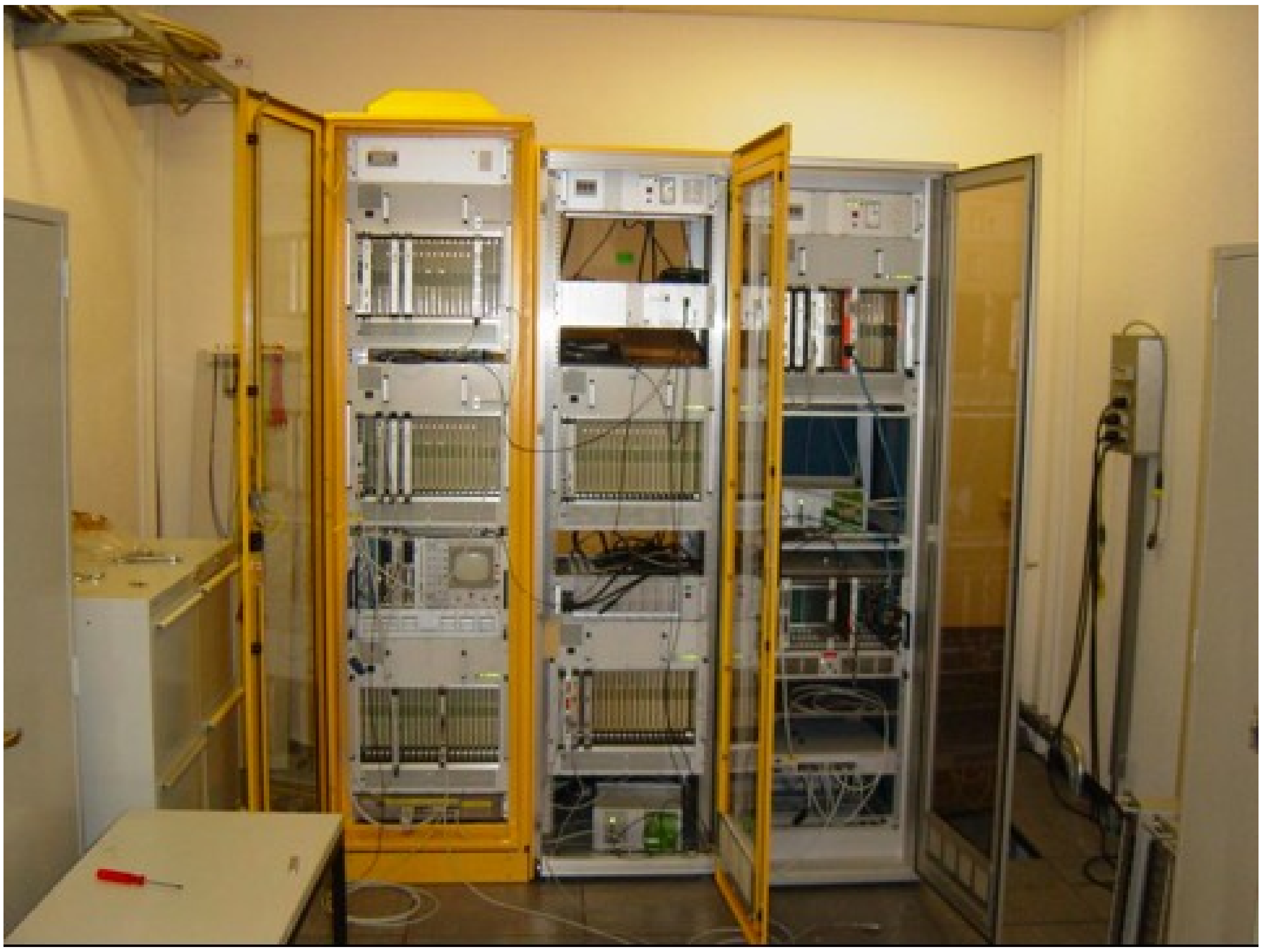
Figure 1.5: Device control layer - VME crates

There are two schemes for designing accelerator control system of the standard model. One is to develop nearly all of the software by themselves, for example, SRRC (Synchrotron Radiation Research Center) in Hsinchu, Taiwan. In the past it was common practice to develop in-house control systems. The other is to make use of professional toolkits, for example, business software SCADA or use one of the available controls software developed specifically for physics experiments, such as EPICS, TANGO, DOOCS and many others. In recent years this approach is common, laboratories and organizations all over the world prefer to collaborate and synergize their efforts on development of common control system software. This approach proved to be very successful[15].

2 Improving the control system reliability

2.1 Reliability Basics

There are several possible definitions of reliability and availability. Common definition of availability as a measure of the ultimate uptime is shown in Equation 2.1:

$$availability = \frac{uptime}{uptime + downtime}$$

Equation 2.1

where *uptime* is the time the system is required to provide its function, and *downtime* is the time the system was not available for operation due to failures or unplanned maintenance. The classic definition of reliability is the probability that a person of system will perform required function under stated conditions for a specified period of time. Therefore time is an essential part of the reliability definition and expected uptime, repair and maintenance times, idle times and unplanned occurrences all affect the measurement of a reliable system.

Repair and maintenance times are necessary to provide a reliable system over a long period of time.

Idle time is defined as time that a system could be providing the required function, but is not. In an accelerator system, this would include times that beam is available, but is not being delivered to a user. Scheduling problems, experimental set ups, other equipment failures can all cause idle time.

Unplanned occurrences will always be a part of complex systems such as accelerators. While they are hard to plan for, they do have to be accounted for in the reliability equation. Power outages, failure of equipment, even weather can cause a system to be unreliable.

We also should take into account that reliability is highly dependent on the type of the desired operation more. If it is short time operation with long maintenance and repair time then it is pretty easy to keep the reliability high. If, however the operation is required for long periods of uninterrupted service with little time of maintenance and repair, reliability will suffer. The simplest representation of availability in terms of reliability parameters MTBF and MTTR is shown in Equation 2.2.

$$availability = \frac{MTBF}{MTBF + MTTR}$$

Equation 2.2

Where MTBF - mean time between failures and MTTR - mean time to repair. And as can bee seen from that equation, general approach to improve the reliability of any system would be to increase MTBF and to decrease MTTR. For highly available systems (99.999% and better) the redundancy becomes essential, because of the non-zero MTTR value. Redundant implementation provides a hot stand-by component, which replaces the failing one as soon as failure is noticed. Redundant systems allow to reduce MTTR to seconds or milliseconds, instead of hours or days in a non redundant system.

Careful monitoring and logging is important in order to achieve high availability. Hardware has a finite lifetime and it in terms of reliability it passes different stages within this lifetime. As it can be seen from the figure there are “burn-in”, normal, and “wear-off” periods. It is crucial to understand where you are on this graph in order to maintain the desired level of availability and properly interpret the data, it is good to remember that it is common for reliability assumptions to presume a constant failure rate[18].

2.2 Reliability of a particle accelerator and control system

Particle accelerators are very complex machines and consist of thousands of components. A typical high energy physics (HEP) accelerator currently has an availability of 75-85%. With so many more components that could potentially fail, the ILC availability would be unacceptably low unless significant attention is paid to component reliability. With the increasing number of components the MTBF decreases and MTTR increases because of the increased complexity of the machine and physical distribution. The study shows that in order to achieve high 75-85% reliability for ILC project, MTBF of some components has to be improved 2-10 times [2]. Therefore reliability considerations when designing a new accelerators play very significant role.

Requirements for the ILC control system are even higher and translate into 99% availability. When we take into account that by design draft ILC control system consists from ~1200 ATCA crates we will get 99.999% availability requirement for each crate. In order to achieve this availability goals single boards running within the crate must have even higher availability of 99.9999% and better. For modern hardware and software solutions this means that Redundancy must be implemented in such places. Otherwise this high numbers of availability are impossible to reach due to long MTTR.

2.3 Components of the CS

Most systems can be broken into four major parts: Hardware, Software, People and Procedures[19]. Accelerator Control system can also be divided in the same manner. Each of these components has its own characteristics and must be looked at in different ways in order to maintain a reliable system.

2.3.1 Hardware

By *Hardware* we understand the *active hardware* components of the system. Such as computers, network devices, power supplies etc. Initially hardware device is supposed to be working and flawless. But when the time passes and under the influence of external forces and natural processes of wear and tear the device or part of it will eventually fail. Repairable devices may be repaired and put back into operation. The ones that cannot be repaired are replaced by spare units. **Thus repairing or replacement of the failing hardware component restores the system to its original not failed state.** For example, if a pump fails, replacing the pump with another similar pump will restore the system to a working state.

Proper commissioning of new equipment ("burn-in") will point out infant mortality problems and incompatibilities, while a good maintenance program can help with equipment as it ages. It is important to keep track of failures of all different components of the system and analyze that data. This will help to understand when and what subsystems reach its wear-off stage and decommission or replace such systems.

2.3.2 Software

Computer software, or just software is a general term used to describe a collection of computer programs that perform some tasks on a computer system.

SOFTWARE components do not age like hardware. The unintended results in software are there from the beginning due to poor design or implementation errors. These unintended results may be “hidden” in the system and the system may function properly for the long periods of time without any visible problem. But from time to time, when certain conditions are met the system fails. These unintended results only occur when all the conditions are met for them to occur, **therefore extensive testing is even more important for software than hardware**.

Unlike hardware, when software becomes unreliable it usually needs to be rewritten, not just replaced with another copy of the same software. All the copies of the same software contain the same errors, thus the new copy will also fail under the same conditions as the original one. In order to fix the software, the software error (“bug”) has to be properly analyzed and studied so that to understand the nature of the problem and to find the design flaws in the software. Therefore it is very important to keep detailed operation log of the system and running software. This information is essential when it needs to recreate the critical conditions in the synthetic environment, either to find the bug, or to make sure that the new version of the software works properly under these conditions.

2.3.3 People

People make mistakes....

No matter how good a person is good at their job, they will make mistakes from time to time. That is human nature. And the goal of reliability engineering for people is to take that nature into account. Proper training and good working environment are important considerations in operator reliability. Human factors such as controls layout, work schedules, even something simple as picking the correct chair can affect the reliability of a system.

General considerations are simple: the environment should help an operator to perform best and to avoid simple mistakes. Interfaces should be natural and easy to use.

2.3.4 Procedures

A procedure is a specified series of actions or operations which have to be executed in the same manner in order to always obtain the same result under the same circumstances (for example, emergency procedures). Less precisely speaking, this word can indicate a sequence activities, tasks, steps, decisions, calculations and processes, that when undertaken in the sequence laid down produces the described result, product or outcome. A procedure usually induces a change. It is in the scientific method.

Clear and easy to follow procedures should be in place in order to achieve high availability. Failure analysis may show that some procedures are not optimal and repeatedly lead to errors. Obviously, in such case procedures have to be redesigned.

3 EPICS Redundant IOC

3.1 Experimental Physics and Industrial Control System

EPICS is a set of software tools and applications which provide a software infrastructure for use in building distributed control systems to operate devices such as Particle Accelerators, Large Experiments and major Telescopes[15]. EPICS software is a good representative of a modern accelerator control software. Even though some flaws and improvements are described in this work, compared to other available control software EPICS is a reliable and powerful software, providing great degree of flexibility and performance.

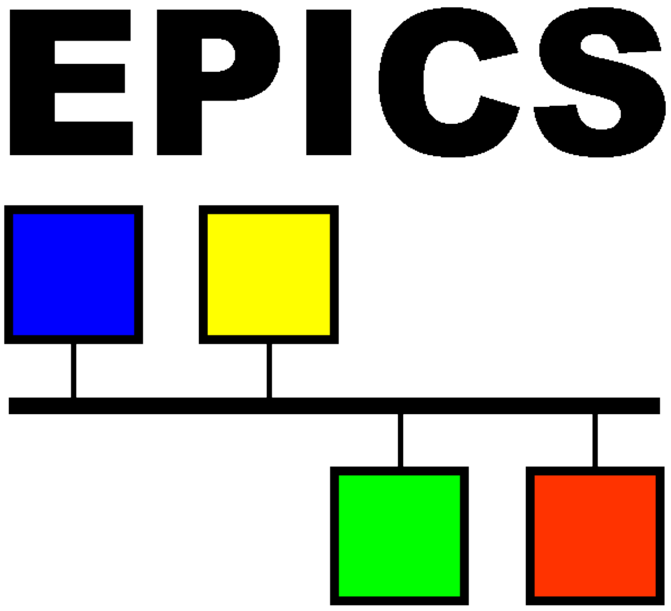

Figure 3.1: EPICS logo

EPICS uses Client/Server and Publish/Subscribe techniques to communicate between the various computers. Most servers (called Input/Output Controllers or IOCs) perform real-world I/O and local control tasks, and publish this information to clients using the Channel Access (CA) network protocol. Figure 3.1 Shows EPICS logo which was inspired by schematic representation of Clients and Server connected via network. Clients and Servers are shown as colored boxes. CA is specially designed for the kind of high bandwidth, soft real-time networking applications that EPICS is used for, and is one reason why it can be used to build a control system comprising hundreds of computers.

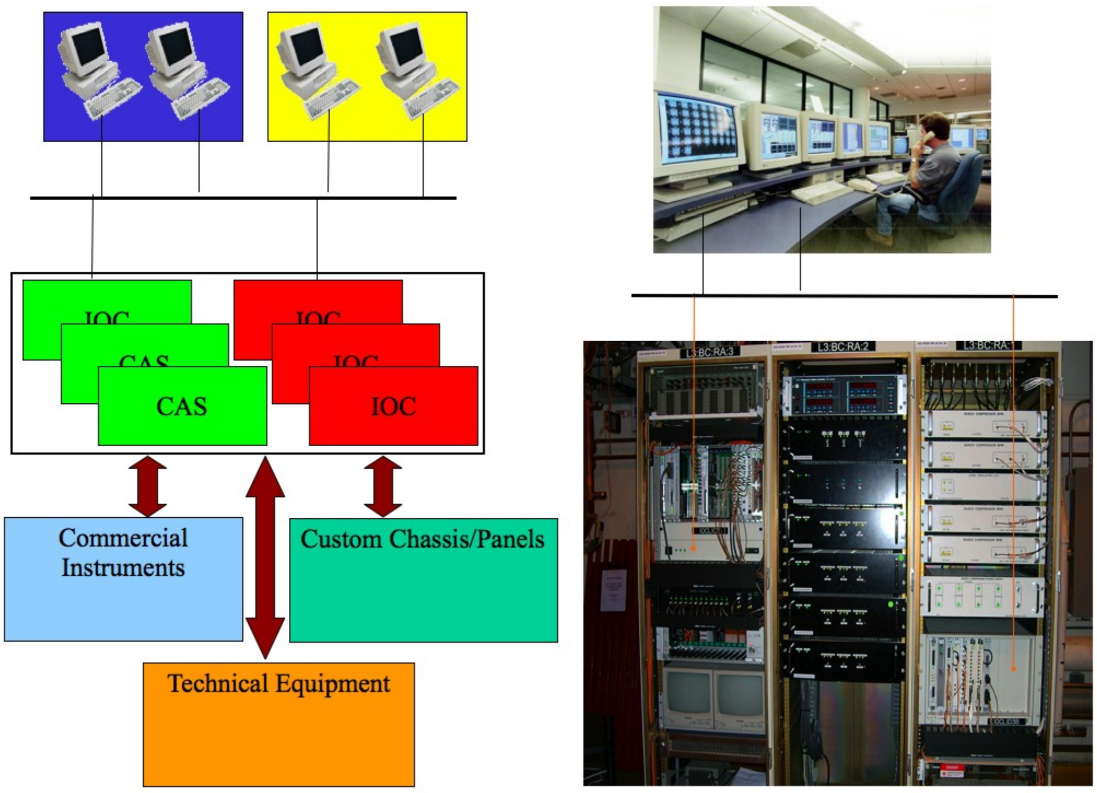

Figure 3.2: EPICS architecture

Originally all EPICS IOCs had to run the vxWorks Real-Time Operating System from Wind River, but since 2004 it has been possible to run IOCs on GNU/Linux, Solaris, MS Windows, MacOS and RTEMS. Portable software is available that allows non-EPICS control systems to act as CA servers. CA clients have always been able to run on a wide range of computers and operating systems — most flavors of Unix, GNU/Linux, Windows, RTEMS and vxWorks. EPICS is also the name of the collaboration of organizations that are involved in the software's development and use. It was originally written jointly by Los Alamos National Laboratory and Argonne National Laboratory, and is now used by many large scientific facilities throughout the world. Development now occurs cooperatively between these various groups, with much sharing of I/O device support and client applications.

EPICS consists of a set of software components and tools with which Application Developers can create a control system (Figure 3.2). The basic components are: OPI: Operator Interface. This is a UNIX based workstation which can run various EPICS tools. IOC: Input Output Controller. This is VME/VXI based chassis containing a Motorola 68xxx processor, various I/O modules, and VME modules that provide access to other I/O buses such as GPIB. LAN: Local area network. This is the communication network which allows the IOCs and OPIs to communicate. EPICS provides a software component, Channel Access, which provides network transparent communication between a Channel Access client and an arbitrary number of Channel Access servers.

Basic Attributes of EPICS

- Tool Based: EPICS provides a number of tools for creating a control system. This minimizes the need for custom coding and helps ensure uniform operator interfaces.
- Distributed: An arbitrary number of IOCs and OPIs can be supported. As long as the network is not saturated, no single bottle neck is present. A distributed system scales nicely. If a single IOC becomes saturated, it's functions can be spread over several IOCs. Rather than running all applications on a single host, the applications can be spread over many OPIs.
- Event Driven: The EPICS software components are all designed to be event driven to the maximum extent possible. For example rather than having to poll IOCs for changes, a channel access client can request that it be notified only when changes occur. This design leads to efficient use of resources as well as to quick response times.
- High Performance: A SPARC based workstation can handle several thousand screen updates a second with each update resulting from a channel access event. A 68040 IOC can process more than 6,000 records per second including generation of any channel access events.

EPICS DATABASE

The heart of an IOC is a memory resident database together with various memory resident structures describing the contents of the database. EPICS supports a large and extensible set of record types, e.g. ai (Analog Input), ao (Analog Output), etc.

Each record type has a fixed set of fields. Some fields are common to all record types and others are specific to particular record types. Every record has a record name and every field has a field name. The first field of every database record holds the record name, which must be unique across all IOCs attached to the same TCP/IP subnet.

A number of data structures are provided so that the database can be accessed efficiently. Most software components, because they access the database via database access routines, do not need to be aware of these structures.

CHANNEL ACCESS

In EPICS, there is a software layer named as CA which connects all clients with all servers. It's the backbone of EPICS and hides all the details of the TCP/IP network from both clients and servers[20]. CA also creates a very solid firewall of independence between all client and server code, so they can run on different processors, and even be from different versions of EPICS. CA mediates different data representations, so clients and servers can mix ASCII, integral, and floating (as well as big- endian and little-endian) types where each uses its natural form.

The design of CA provides very high performance of allowing throughput rates on the order of 10,000 "gets" or "puts" per second under heavy load, yet minimizing latency to about 2 milliseconds under light load. If the medium allows it, many clients and servers can simultaneously sustain these rates. Since EPICS is a fully-connected and flat architecture, every client and every server make connections with no 'relay' entities, so there are no bottlenecks beyond the physical limits of the medium. CA also uses a technique called 'notify by exception' or callback (also called "publish and subscribe"). Once a client has expressed an interest in certain data to a server, the server notifies the client only when the data changes. This not only minimizes traffic, but signals both the health of the server and the freshness of the data. With CA protocol, all data carry time-stamps, validation information based on both the quality of connection, and validity down to the hardware layer as explained later. Thus, a critical client implementing a global feedback loop can assure it is operating only with fully validated data[21].

In summary, EPICS provides a software toolkit for implementing control systems following the `standard model' paradigm. Scientific labs, industrial partners, and other users augment the toolkit. EPICS control systems can achieve modularity, scalability, robustness, and high speed in hardware and software, yet remain largely vendor and hardware-independent. EPICS provides seamless integration of several data acquisition bus standards. The software development environments for intelligent local controllers and workstations can be identical. Good documentation, training, and support are available. Standard systems can be configured with text editors and other simple tools, yet full customization is available to sophisticated sites.

3.2 Redundant IOC introduction

As it was explained in the previous chapter, redundancy is one of the key techniques used to achieve high reliability. Until recently EPICS based control system did not have a support for redundancy at all. Therefore the sites that do require redundancy had to use third party implementations. For example DESY has been using redundant controllers for more than 20 years. The experience with commercial implementations and the requirements for the new XFEL (x-ray free electron laser) project were the main driving forces for a new development of redundant IOC's in the EPICS environment[9]. Two major fields of application were defined:

- Redundancy for cryogenic plants. In this case failures can be caused by malfunctioning hardware components like power supplies, fans, CPU boards or communication links. Over the years it was sometimes necessary to manually switch mastership between processors because some maintenance work had to take place during a runtime period. (runtime periods typically last for one year or more) Another case where manual switching of mastership was proven to be useful was due to major software changes. While this operation can be avoided in the current commercial implementation by issuing online changes to the database etc. – it is important for the EPICS case where online add and delete of records or databases is not available in due time.
- Redundancy for controllers in the XFEL tunnel. While the switching in the first case is mainly caused by manual action, we expect this to happen automatically in the XFEL tunnel. Here we expect radiation damage to the equipment like memory and CPU. Loading new programs is probably not be as critical as in the first case because of more frequent maintenance days which would allow for such operations. As a consequence the processors must be disjoint from each other by at least one hundred meters. The fail over methods and the error detection of malfunctioning processors must concentrate on this case.

By the design draft one major goal was set: Any redundant implementation must make the system more reliable than the non-redundant one. Precaution must be taken especially for the detection of errors that shall initiate the fail-over. This operation should only be activated if there is no doubt that maintaining the actual mastership definitely causes more damage to the controlled system than an automatic fail-over. The fail-over time in any case was defined to be more than several seconds and less than 15s[22]. The system implemented shows failover times around 3s.

Originally, the project was intended to support only vxWorks and the source code was very specific to it. However, later it was observed that support for other operating system is desirable. Here at KEK we use a software-IOC on Linux which functions as a "gateway" from an old control system to the EPICS-environment. In addition, for the ILC project ATCA-based systems under Linux control will be used. Redundant IOCs are highly desirable for this project. Thus, the redundant IOC has been ported to EPICS libCom/osi; this implies that the current implementation should work on any EPICS-supported OSs[10].

3.3 Redundant IOC architecture

3.3.1 Hardware Architecture

The hardware architecture consists of two redundant IOCs controlling a remote I/O via shared media such as the Ethernet (see Figure 3.3). The redundant pair shares two network connections for monitoring the state of health of their counterpart, where the private network connection is used to synchronize the backup to the primary and the global network is used to communicate data from the primary to any other network clients requiring the data[9].

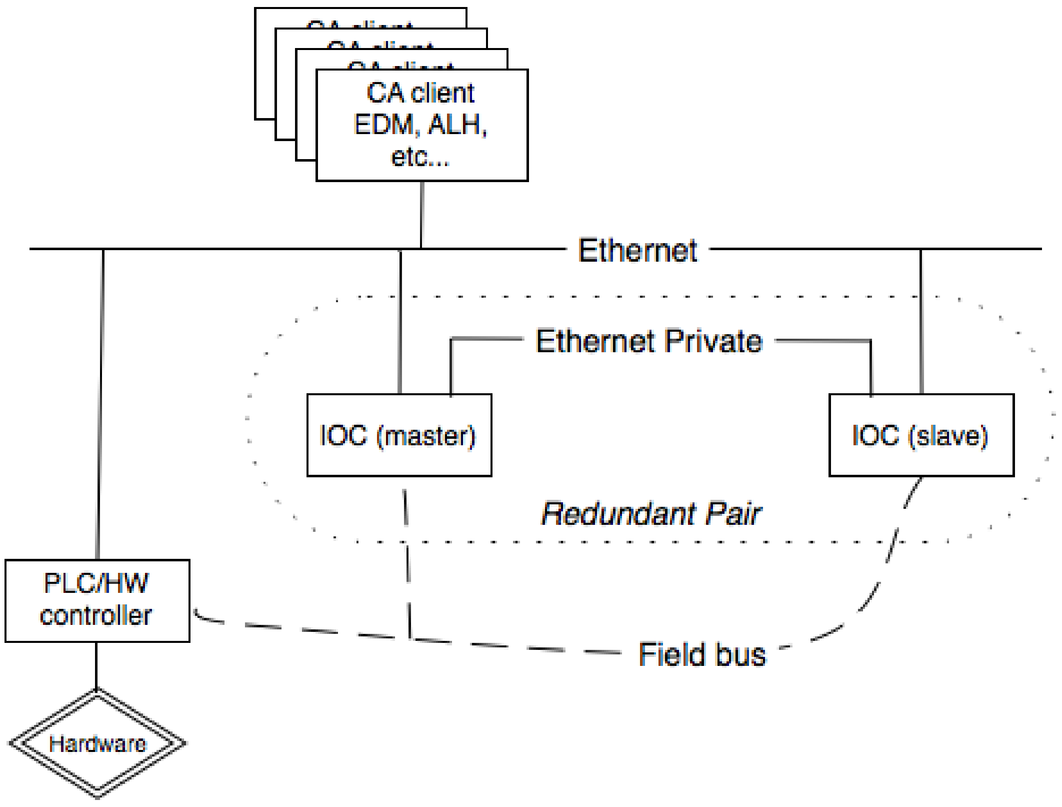

Figure 3.3: Redundant IOC hardware architecture

3.3.2 Software Components

An EPICS redundant IOC consists from four major parts Redundancy Monitoring Task (RMT), Continuous Control Executive (CCE), State Notation Language Executive (SNLE)[22] and IOC-part (which is same as non-redundant IOC). RMT – is a key component of that system. RMT is responsible for monitoring all other parts of the system, checking connectivity and making decisions regarding fail-over. The CCE is responsible for database synchronization between peers. The SNLE is responsible for synchronization of State Notation Procedures between peers. IOC part is almost the same as in the original EPICS IOC, with a few modifications that allow to control database scanning threads and channel access server thread form RMT. Since version 3.14.10 the base distribution of the EPICS software already includes all the needed changes for the Redundant IOC, therefore no modification is done to the "original IOC" part.

EPICS Redundant IOC

Redundant software, such as EPICS IOC, may have internal state that must be preserved between fail-overs and switchovers. Depending on the particular software corresponding RMT-driver has to be implemented. It is driver,s responsibility to do all the synchronization, locking and verification. RMT is only informing its drivers about the state of the unit and its partner and gives the control commands such as *start, stop, test i/o etc.* In IOC's case database records and SNL programs have to be synchronized between master and slave peers. Therefore CCE and SNLE components are implemented to perform this functionality. Within IOC there are some parts, that are needed to be controlled from the RMT depending on the state of the unit. For example on the master unit channel access server should be up and running, but on the slave unit it should not. Same thing with database scanning threads. Database is copied from the master to the slave unit, thus IOC's database update mechanisms should be stopped on the slave unit.Thus, CCE, SNLE, database scan threads and channel access server are controlled by RMT and are called "RMT drivers" accordingly. Any other software which is need to be redundant has to implement RMT-driver API interface , that is described further in this chapter. RMT itself is implemented as a state-machine with the following states[22] (See Figure 3.4):

- Initial state
- Master
- Slave
- CMD to Fail
- Master I/O Fail
- Slave I/O Fail

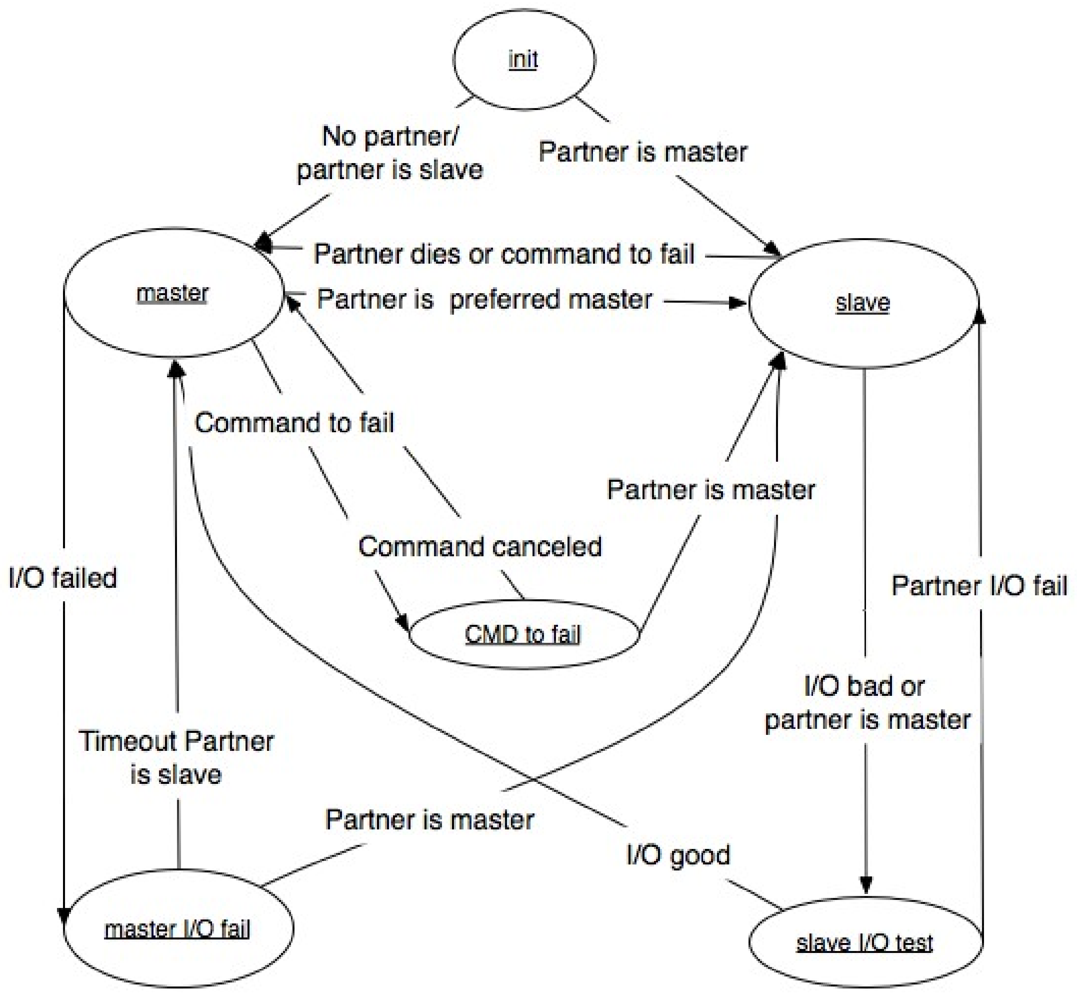

Figure 3.4: RMT state transition diagram

3.3.3 RMT API

Redundancy Monitoring Task defines current state of the node and communicates with its partner. As for the IO drivers (and other parts of the system, that RMT has the control of) there should be some mechanism to control them. For that reason RMT API was defined as an interface between the RMT and software components in the IOC which have to be controlled by the RMT. A component can be a major part of the EPICS IOC like CCE, SNLE or some IO-driver. All these components share the same interface to the RMT. Some components may be much simpler than the others, therefore the component may implement the RMT API only partially. The interface must be implemented as functions defined in the component and callable by RMT. The addresses of these functions are the entry table to the component. During the initialization, the component has to check whether its running within Redundant IOC. In case of redundancy it has to call rmtRegisterDriver() with the address of the entry table as an argument. Then the component goes to the stopped state and from now its RMT's responsibility to call the corresponding functions to start and stop the component. In case of non-redundant operation the component works normal (start). This allows to use the same code for redundant and non-redundant operation.

The RMT API defines the following functions:

- start: Get access to the IO and start processing.
- stop: Do not access the IO and stop processing.
- testIO: Initiates a procedure to test access to the IO. This function may be used by IO drivers. getStatus: Get status of the driver.
- shutdown: This function is called before the IOC is rebooted. It terminates transient activities, deactivates interrupt sources and stops all driver tasks.
- getUpdate: This routine tells the component to get an update from the redundant IOC. It is normally called by the RMT on the inactive IOC.
- startUpdate: This routine tells the component to start updating data from the redundant IOC (monitoring). It will first read all fields (depending on the mode) from the redundant partner. It is normally called by the RMT on the inactive IOC.
- stopUpdate: This routine tells the component to stop updating data from the redundant counterpart. This routine is normally called by the RMT on the inactive IOC.

The RMT can call these functions using an entry table that is transferred from the component to the RMT during initialization[23].

The CCE and the SNL executive are two such RMT-controlled components and they implement the RMT-driver interface mentioned above. Other components may be IO drivers, or any other piece of software. For example, the channel access server (RSRV) in redundant IOC implementation is one of the RMT-controlled components and it implements the RMT-driver interface.

EPICS Redundant IOC

Both standalone and IOC-related software such as device drivers can be made redundant using RMT. One of the functions of RMT is to check status of its drivers and exchange this information with its partner. Each RMT driver implements its own logic for checking whether it is “OK” or “NOT OK” depending on the particular driver and implementation. The drivers and hardware that do have some internal state, needed to be preserved have to implement their own synchronization mechanisms. RMT and EPICS redundant IOC does not provide general mechanisms for synchronization and locking. There are software environments created to ease the development of redundant and highly reliable systems. Such software may be adopted for the usage with the EPICS Redundant IOC. See chapter about ATCA for more discussion on that topic.

3.4 Redundant IOC generalization: bringing RIOC to other architectures

The initial development and implementation of the EPICS redundant IOC was done for vxWorks operating system only. This resulted in the software that was unusable on any other operating system other than vxWorks. However there was a lot of interest in redundancy on other operating systems. The Redundant IOC developed at DESY seemed like a good basis to create more general software, that could be used in wide set of application. An example of such application is channel access gateway (caGateway). We wanted to make it redundant and we were looking for redundancy controllers for that project. Redundancy Monitoring Task seemed to fit very well with the only one problem. RMT worked on vxWorks only, and caGateway worked only on Linux. Therefore I joined the development of the redundant IOC at DESY with the goal to implement multi-platform EPICS redundant IOC.

3.4.1 Porting process

The original EPICS redundant software was written in very wxWorks dependent style. There was a problem in front of me, how to make that software to support Linux, vxWorks and possibly other operating system. The common approach to create multi-platform software is to use additional “wrapping” API. EPICS is a multi-platform application too and it uses the same approach. EPICS uses so called libCom/OSI (Operating System Independent Library) as a “wrapping” API. libCom/OSI is a part of EPICS project and it was introduced in version 3.14. EPICS had been vxWorks only software too for many years, until version 3.14 that brought support for multiple operating systems. LibCom/OSI was developed to hide the operating system differences for the most common functions used in EPICS software. Therefore libCom/OSI was a perfect candidate to be used to make Redundant IOC operating system independent.

The process of porting required all vxWorks specific function calls to be replaced with the EPICS libCom/OSI (Operating System Independent library). During the porting process some serious errors were found and fixed in the Redundant IOC code. vxWorks is a real-time system and Linux is not, those two systems have very different process models as well. vxWorks runs all the processes in real-mode, all the processes share the same memory address space and there is no memory protection mechanism. Therefore memory access errors may be not so easy to find on vxWorks, because writing to some other "process" memory does not result in "segmentation fault" or similar system error. The CCE software had several such bugs, but on vxWorks they were not visible during normal testing conditions. But as far as those bugs were present there, they might have presented themselves under certain conditions. As soon as the software was ported to libCom/OSI and run on Linux operating systems many "segmentation fault" error appeared. Careful investigation of the memory core dumps and source code analysis allowed me to find and fix those bugs. Eventually Redundant IOC run successfully on Linux machine. Testing and evaluation was done to run the generalized version of Redundant EPICS IOC on vxWorks to ensure that no functionality was broken during the porting process.

In order to accomplish the porting of the Redundant EPICS IOC the following new OSI functions were introduced:

- **epicsMutex.h:** epicsMutexLockWithTimeout()
- **epicsMutex.h:** epicsMutexOsdLockWithTimeout()
- **epicsThread.h**: epicsThreadDelete()
- **epicsTime.h:** epicsTimeGetTicks ()[1]

One of the goals of the porting was to create general software for redundancy, that could be used not only with the EPICS-related software. Therefore divided and packaged the generalized version of Redundant IOC into several independent libraries: rmtLib, cceLib. Depending on the application requirements it is now possible to include/exclude the needed functionality with simple editing of the Makefile.

Further, some modification of the EPICS base were done to implement CCE hooks and the RMT-driver interface to the CA server (RSRV), and in database scan tasks. These modifications were also made operating system independent. The following is the list of affected source files:

- base/src/db/dbAccess.c
- base/src/db/dbScan.c
- base/src/dbStatic/dbBase.h

1 These modifications were not included into the base distribution, some of them were incorporated as a part of the RIOC distribution, while others were replaced by alternative solutions. See the discussion related to redundancy in EPICS tech-talk mail list for the details.

- base/src/dbStatic/dbLexRoutines.c
- base/src/rec/aiRecord.dbd
- base/src/rsrv/camsgtask.c
- base/src/rsrv/online_notify.c
- base/src/rsrv/cast_server.c
- base/src/rsrv/caservertask.c

(see Appendix for details)

Those modifications[2] were sent to the core developers of the EPICS system and later resulted in the support for Redundancy in EPICS base distribution. This functionality was introduced in version 3.14.10[11].

3.4.2 Results and performance

The generalized version of the redundant IOC was successfully used on vxWorks, Linux, Mac OS X, and Solaris. The porting process allowed to reveal several serious bugs in the original code. The generalized software was tested for performance issues on Linux platform. The computers used for the tests were regular PC-compatible computers running Red Hat Fedora 5 linux system. Tests showed that the system synchronization speed limit was around ~5000 records/second for the PC with Intel Pentium 4 3GHz 1x core, 2x 100 Mbit Ethernet cards; functioning solely as a redundant IOC.

3.4.3 Failover Performance

In my tests the failover time was in range of 0.3-3 seconds. The following is the event log of the slave peer, you can see the event when manual fail over command is issued.

2 These modifications were not included into the base distribution, some of them were incorporated as a part of the RIOC distribution, while others were replaced by alternative solutions. See the discussion related to redundancy in EPICS tech-talk mail list for the details.

```
0057 2009-07-17 09:44:25.646 'RMT','synchronisation state changed InSync=1'
0058 2009-07-17 09:45:24.925 'RMT','LS=SLAVE, RS=CmdToFail, PuEth=10, PuUP=1, PrEth=9,
PrUP=1, DrR=0, DrF=0, DrIS=9, DrNiS=0, InSync=1'
0059 2009-07-17 09:45:25.246 'CCEXEC','PRR StopUpdate(0)'
0060 2009-07-17 09:45:25.246 'RMT','Partner ist CmdToFail -> switch to MASTER'
0061 2009-07-17 09:45:25.246 'RMT','LS=MASTER, RS=CmdToFail, PuEth=10, PuUP=1, PrEth=9,
PrUP=1, DrR=0, DrF=0, DrIS=9, DrNiS=0, InSync=1'
0062 2009-07-17 09:45:25.649 'scan0.2','PRR started'
0063 2009-07-17 09:45:25.655 'scan5','PRR started'
0064 2009-07-17 09:45:25.655 'scan0.5','PRR started'
0065 2009-07-17 09:45:25.656 'scan0.1','PRR started'
0066 2009-07-17 09:45:25.656 'CAS-TCP','PRR started'
0067 2009-07-17 09:45:25.656 'CCEXEC','PRR started'
0068 2009-07-17 09:45:25.657 'scan10','PRR started'
0069 2009-07-17 09:45:25.657 'scan2','PRR started'
0070 2009-07-17 09:45:25.657 'scan1','PRR started'
0071 2009-07-17 09:45:26.593 'RMT','LS=MASTER, RS=SLAVE, PuEth=10, PuUP=1, PrEth=9,
PrUP=1, DrR=9, DrF=0, DrIS=0, DrNiS=9, InSync=1'
0072 2009-07-17 09:45:27.455 'RMT','synchronisation state changed InSync=0'
          LS=MASTER, RS=SLAVE, PuEth=10, PuUP=1, PrEth=9, PrUP=1, DrR=9, DrF=0, DrIS=0,
DrNiS=9, InSync=0epics>
```

With the bold font I marked the time when the switch-over process begins and ends consequently. The time difference between this two points is: 0.732 second, this is a typical switch over time.

And another switchover example, with the same configuration

```
0076 2009-07-17 09:49:33.264 'RMT','LS=SLAVE, RS=CmdToFail, PuEth=9, PuUP=1,
PrEth=8, PrUP=1, DrR=0, DrF=0, DrIS=9, DrNiS=0, InSync=1'
0077 2009-07-17 09:49:34.772 'CCEXEC','PRR StopUpdate(0)'
0078 2009-07-17 09:49:34.772 'RMT','Partner ist CmdToFail -> switch to MASTER'
0079 2009-07-17 09:49:34.772 'RMT','LS=MASTER, RS=CmdToFail, PuEth=9, PuUP=1, PrEth=8,
PrUP=1, DrR=0, DrF=0, DrIS=9, DrNiS=0, InSync=1'
0080 2009-07-17 09:49:34.920 'RMT','LS=MASTER, RS=SLAVE, PuEth=9, PuUP=1, PrEth=8,
PrUP=1, DrR=0, DrF=0, DrIS=9, DrNiS=0, InSync=1'
0081 2009-07-17 09:49:36.312 'CCEXEC','PRR started'
0082 2009-07-17 09:49:36.313 'scan10','PRR started'
0083 2009-07-17 09:49:36.313 'scan5','PRR started'
0084 2009-07-17 09:49:36.314 'scan2','PRR started'
0085 2009-07-17 09:49:36.314 'scan0.5','PRR started'
0086 2009-07-17 09:49:36.315 'scan0.2','PRR started'
0087 2009-07-17 09:49:36.315 'scan0.1','PRR started'
0088 2009-07-17 09:49:36.315 'scan1','PRR started'
0089 2009-07-17 09:49:36.316 'CAS-TCP','PRR started'
0090 2009-07-17 09:49:38.108 'RMT','synchronisation state changed InSync=0'
          LS=MASTER, RS=SLAVE, PuEth=9, PuUP=1, PrEth=8, PrUP=1, DrR=9, DrF=0, DrIS=0,
DrNiS=9, InSync=0epics>
```

Switch-over time: 3.052 second. This is one of the worst cases. In case of a automatic failover here is the transcript:

```
0146 2009-07-17 09:54:33.314 'PRIVETHN','private ethernet is down'
0147 2009-07-17 09:54:33.314 'RMT','LS=SLAVE, RS=UNKNOWN, PuEth=9, PuUP=1, PrEth=8,
PrUP=0, DrR=0, DrF=0, DrIS=9, DrNiS=0, InSync=1'
```

```
0148 2009-07-17 09:54:33.314 'CCEXEC','PRR StopUpdate(0)'
0149 2009-07-17 09:54:33.315 'PUPLETHN','public ethernet is down'
0150 2009-07-17 09:54:33.324 'RMT','Communication lost on both channels -> switch to
MASTER'
0151 2009-07-17 09:54:33.324 'RMT','LS=MASTER, RS=UNKNOWN, PuEth=-1, PuUP=0, PrEth=-1,
PrUP=0, DrR=0, DrF=0, DrIS=0, DrNiS=9, InSync=1'
0152 2009-07-17 09:54:34.875 'CCEXEC','PRR started'
0153 2009-07-17 09:54:34.876 'scan10','PRR started'
0154 2009-07-17 09:54:34.876 'scan5','PRR started'
0155 2009-07-17 09:54:34.878 'scan2','PRR started'
0156 2009-07-17 09:54:34.878 'scan0.5','PRR started'
0157 2009-07-17 09:54:34.879 'scan0.2','PRR started'
0158 2009-07-17 09:54:34.879 'scan0.1','PRR started'
0159 2009-07-17 09:54:34.879 'scan1','PRR started'
0160 2009-07-17 09:54:34.880 'CAS-TCP','PRR started'
0161 2009-07-17 09:54:34.992 'RMT','synchronisation state changed InSync=0'
          LS=MASTER, RS=UNKNOWN, PuEth=-1, PuUP=0, PrEth=-1, PrUP=0, DrR=9, DrF=0,
DrIS=0, DrNiS=9, InSync=0epics>
```

Time required for switchover since the partner was noticed missing is: 1.566 second. To measure “real” time lost in the switchover I implemented the following test. The calc record is being scanned every second and it is incremented by 1. When the switchover happens, it is likely that some cycles will be dropped and the expected value and the value received after the switch over will be different. It can be clearly seen from the following pictures data and pictures. The first column is time and the second column is the variable value. Figure 3.5 and Figure 3.6 show the change of the process variable. The gap in the middle is 30 second Channel Access timeout; after the gap the client is reconnected to the stand-by IOC and the value is received from there.

```
00.898630 1523
01.900604 1524
02.902560 1525
03.904522 1526
04.906492 1527
05.908453 1528
06.910414 1529
07.912377 1530
08.914340 1531
09.916303 1532
10.918278 1533
11.920230 1534
12.922198 1535
50.900448 1570
51.902407 1571
52.904357 1572
53.906313 1573
54.908271 1574
55.910227 1575
56.912179 1576
57.914133 1577
58.916090 1578
59.918043 1579
```

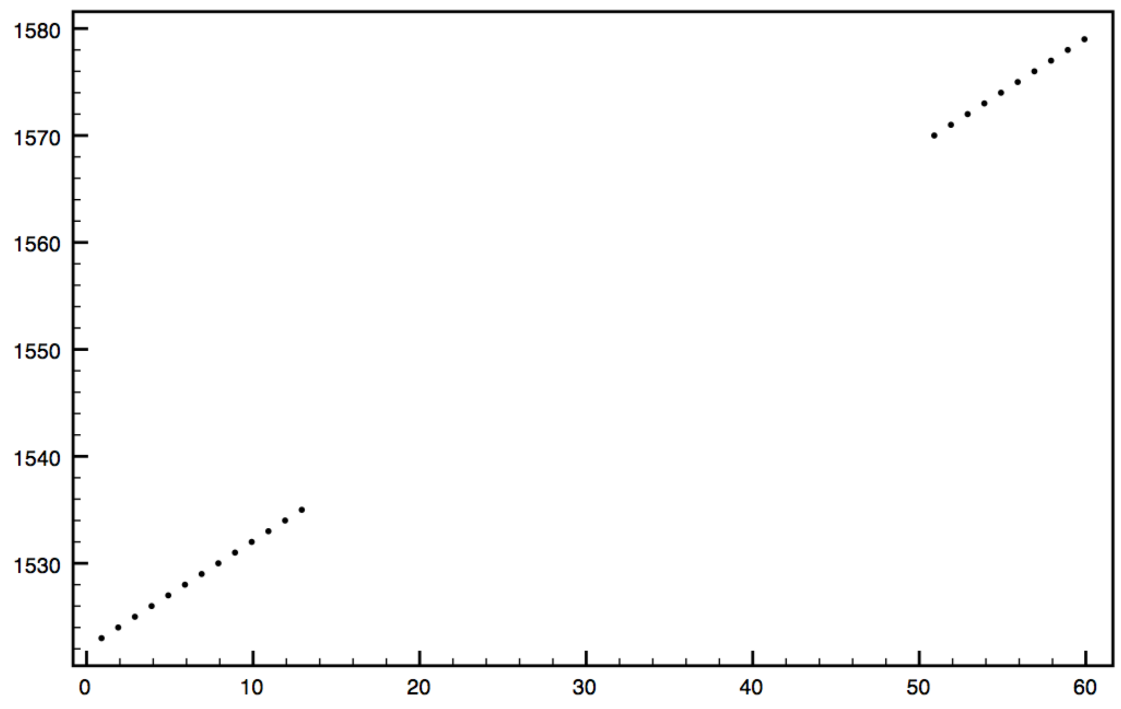

Figure 3.5: Failover time measurement

The calculated time loss is 2.904 second.

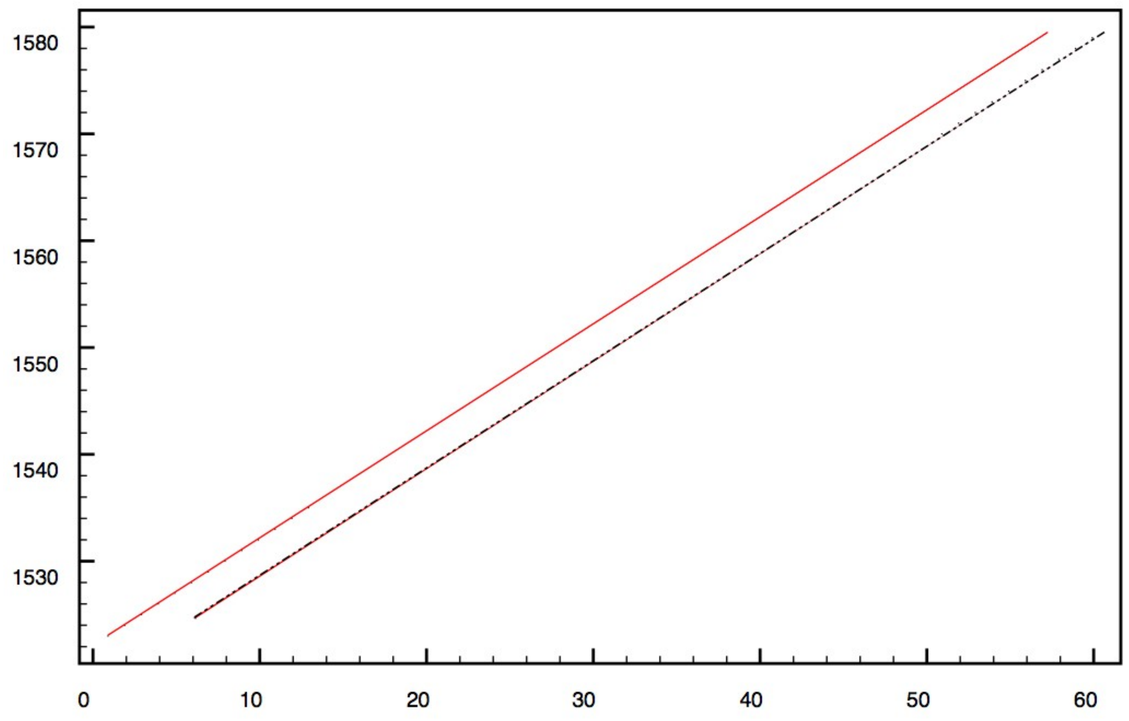

Figure 3.6: Expected and received values

3.4.4 Conclusion

The generalization of the Redundant IOC to libCom/OSI and extensive testing of the software during the porting process eventually led to inclusion of the support for Redundancy in the EPICS base distribution. This work had a great influence on Redundant IOC software to reach production state. The generalized version can be now used to provide redundancy for software not being EPICS IOC. The next 2 chapters describes one of the application developed with the generalized version of Redundant IOC.

4 redundant Channel Access Gateway

4.1 Channel Access Gateway description

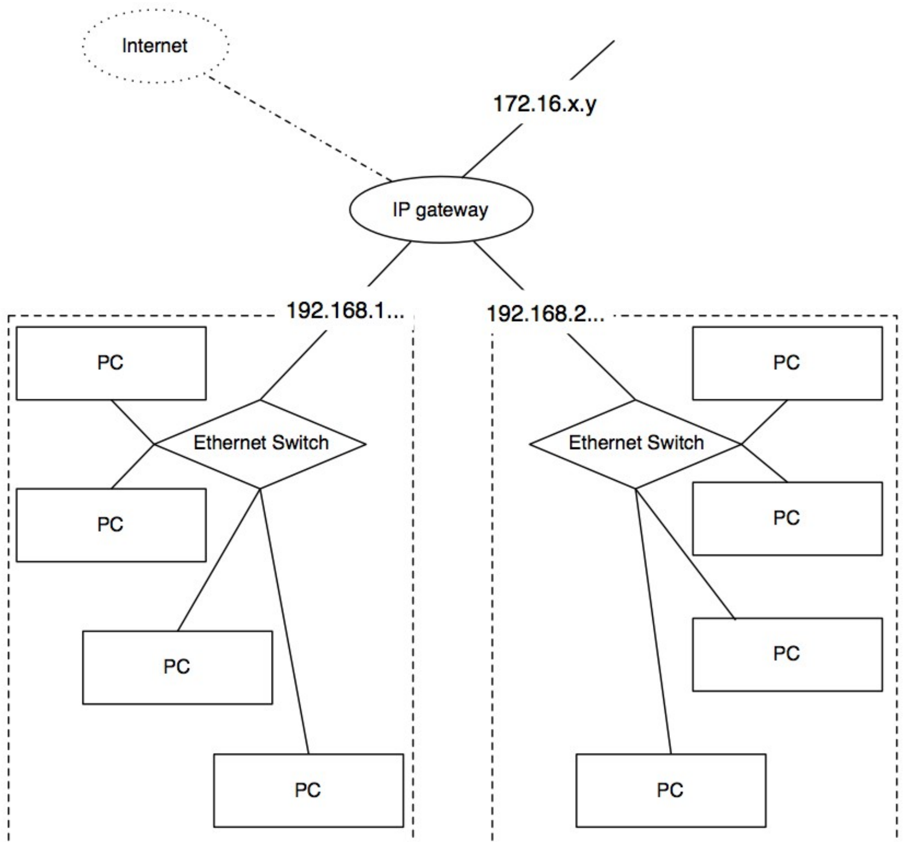

Figure 4.1: LAN with several subnetworks

Channel Access (CA) Protocol works on top of TCP/IP stack. For its discovery purposes channel access uses broadcast capability of TCP/IP. Therefore discoverability of an EPICS IOC is limited by the borders of a "broadcast-domain", which is usually equal to size of a network subnet. Nevertheless modern LANs usually consist from more than one network subnets (Figure 4.1). Sometimes that division is done for administrative and security reasons and sometimes the networks grow too big. In any case the broadcast packets cannot (or better say should not) cross between two subnets. That configuration prevents broadcast storms and it is good for network throughput. However it creates some problem for protocols that rely on broadcast functionality and channel access is one of them.

redundant Channel Access Gateway

For example if there are several subnets in a lab, and there are EPICS IOCs and EPICS clients spread across different subnets, then some of these IOCs and clients cannot find each other. When CA client needs to find some specific data (which is called "process variable" in EPICS terminology - PV), the client sends UDP broadcast with the request for the need process variable. When the IOCs receives that broadcast it searches its own database for the presence of the required PV and if it is found the IOC sends the reply to the client. Then client connects to the IOC. But if the IOC belongs to another subnet, then the searching broadcast cannot reach the IOC. To resolve this problem channel access gateway is used. An example of such usage is shown on Figure 4.2 CA Gateway Usage at KEKB and Linac.

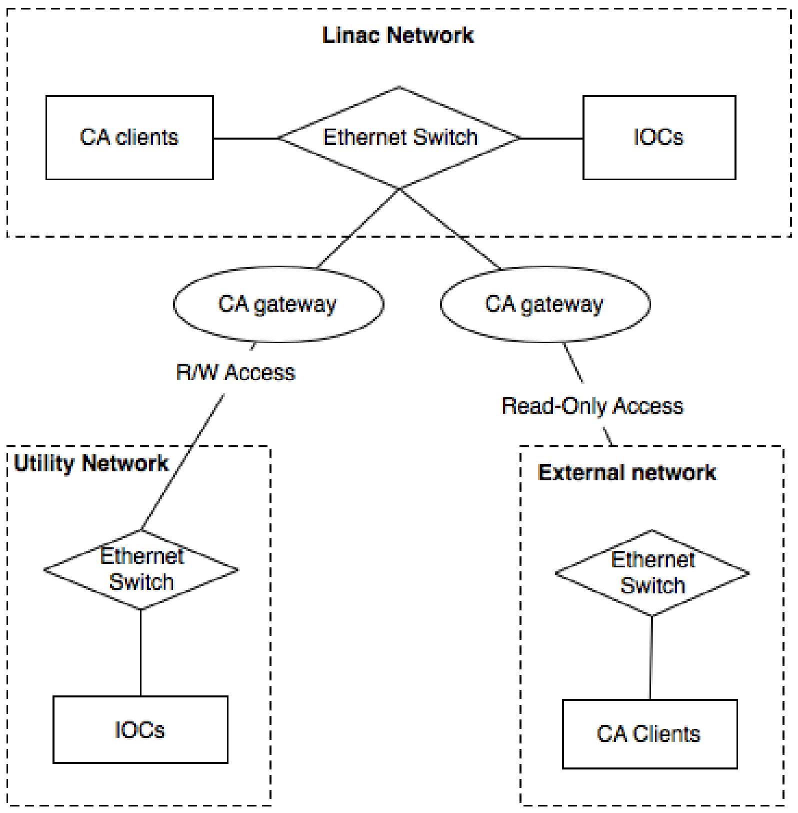

Figure 4.2: CA Gateway Usage at KEKB and Linac

redundant Channel Access Gateway

4.1.1 Channel Access Gateway operation

CA Gateway (CAG) can be seen as Channel Access server on one subnet (A) and as Channel Access client on another subnet (B). There are some clients on subnet A, that need to get connected to CA servers on subnet B. Let's say that we have a client CAC1 on subnet A, that wants to access a process variable named PV1. Process variable PV1 is served by Channel Access Server CAS1 on subnet B. So let's how CA Gateway works.

First CAC1 sends a broadcast request for PV1 on subnet A. CAG receives that request and looks its internal database for that name. If that name is unknown to CAG it puts it into the database and issues a broadcast request for PV1 on subnet B. CAS1 replies to CAG, because it has that variable and thus CAG updates the its internal record that PV1 is available on subnet B and served by CAS1. Next time the request for PV1 comes, it will not search for it, but directly connect to CAS1. Anyway, after the search process is finished, CAG sends a reply back to CAC1 stating that PV1 is available on CAG. Next step CAC1 connects to CAG server side and CAG client side on subnet B connects to CAS1. So CAG works as "pipe" from subnet B to A.

But what happens if CAS1 will look for a PV2 that is not provided by any servers on networks A and B. Then there will be a record in CAG database that will show that and next time the reply comes for PV2, CAG will not send a broadcast request on subnet B.

4.1.2 Channel Access Gateway -> single point of failure!

Channel Access Gateway is a software that forwards broadcasts between the subnets and helps Channel Access clients and servers to find each other on different subnets. That architecture implies that CA gateway becomes the single point of failure. When the CA gateway stops working all the subnets it was connecting become unreachable to each other. This makes the CA gateway the perfect candidate to implement redundancy. The redundant ca Gateway eliminates the single point of failure and reduces the repair time drastically.

Therefore the redundancy for ca gateway was sought. Channel Access gateway does not have any internal state that needs to be synchronized within redundant pair. This simplifies the implementation. Still the redundancy controller is needed, because if two gateways start to run on the same networks they will create problems for the CA. Two gateways will both replicate the broadcasts from the clients and then they will both replicate the reply from the server. This kind of behavior disturbs the consistency of the CA network. Therefore Redundancy controller is essential to manage the status of the CA Gateways.

As it was described in the previous chapter, Redundant IOC consists from the Redundancy Monitoring Task and other components. The RMT plays the role of the Redundancy Controller. The generalized version of RMT can be used on Linux (and other operating systems supported by EPICS lib/OSI) to make any software redundant. The RMT performed very well as a redundancy controller for EPICS Redundant IOC, and because my involvement in the project, the RMT was chosen as a Redundancy Controller software to implement Redundant Channel Access Gateway.

4.2 Redundancy without load-balancing

The first and the main goal was to implement redundant Channel Access Gateway. The Redundant CA Gateway should perform as “non-redundant” CA Gateway in terms of functionality. The implementation is a little bit different on the other hand.

4.2.1 Redundant Channel Access Gateway Architecture

As the EPICS Redundant IOC, redundant Channel Access gateway consist from the “redundancy pair” in this case it is two computers running Redundant Channel Access Gateway software on them. Both of these computers have two Ethernet interfaces, those interfaces belong to different subnets - the subnets that CA gateway is supposed to interconnect.

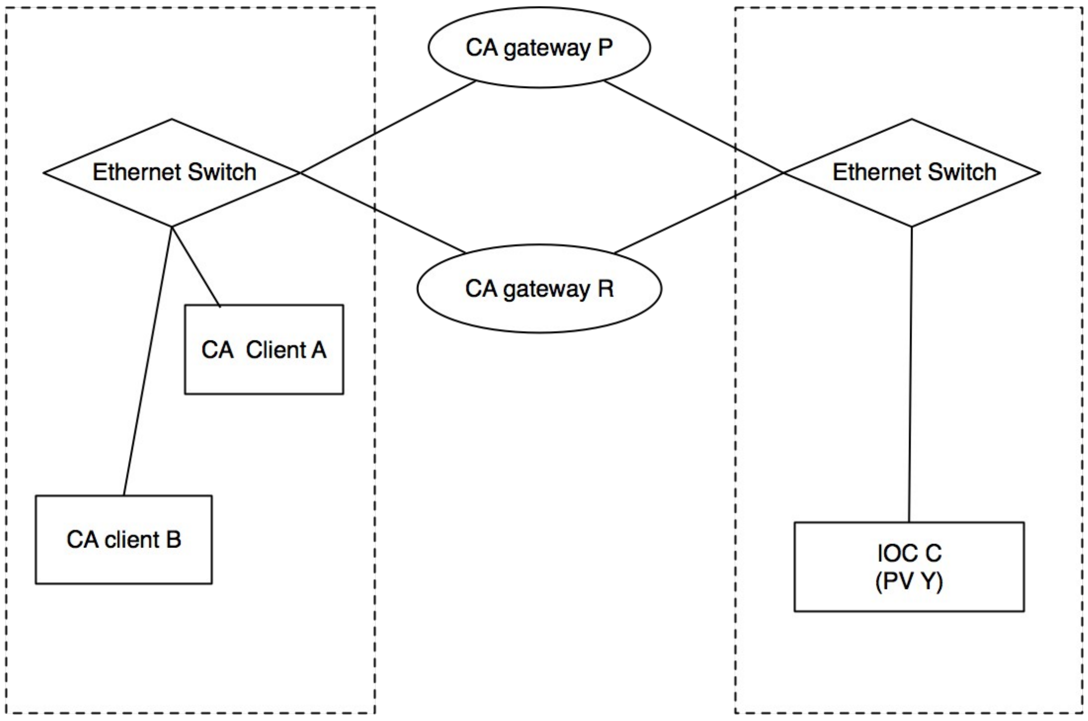

Figure 4.3: Redundant CA Gateway network architecture

At any given point of time only one of the gate should forward the broadcast requests and replies. The other gateway has to standby and come into operation in case of the failure of the first one. If we just run two gateways with identical configuration on two same subnets, then the replies and search broadcasts will be always doubled (see Figure 4.4 and Figure 4.5). This makes and inconsistency problem for Channel Access Clients. In Channel Access, each process variable should have unique name within one broadcast domain. Therefore two gateways will interfere the normal operation of the Channel Access. For that reason the standby ("slave") gateway should not send any replies to the channel access clients.

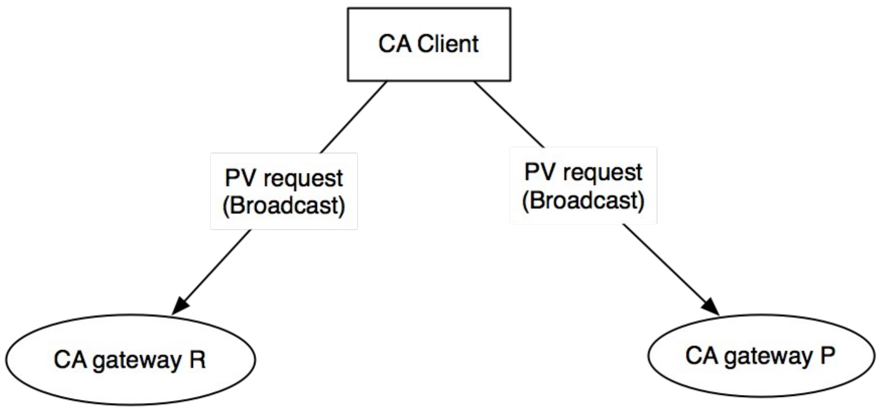

Figure 4.4: PV request reaches both CA gateways

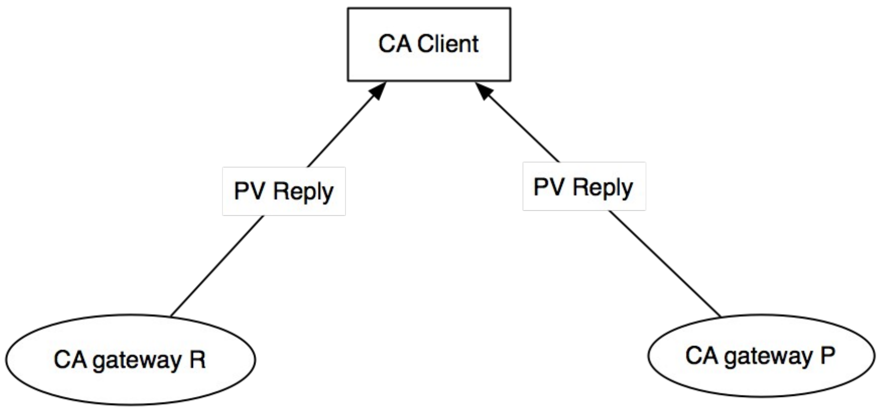

Figure 4.5: Both CA Gateways send reply to the CA client

When I was designing this software, I had several options solve this problem. One way is link RMT and CA Gateway into one binary and use RMT API to control CA Gateway. This approach requires to modify the source code of the CA Gateway, which is not a bad thing by itself, but potentially it may introduce some bugs. Taking this point under consideration and having another way of implementing CA Gateway I chose the latter one, that is described further.

CA Gateway does not require any synchronization between peers within “Redundancy Pair”, therefore it does not really need to be “fully” controlled by the RMT. All we need is to have it running and serve client requests when it is needed (“master mode”). And when it is not needed (“slave mode”), the client request must not be served. UNIX operating system paradigm is to use small building blocks, each is doing some small task; and use those blocks to solve some sophisticated tasks. That approach I chose to solve my problem. Every modern UNIX-like system has IP firewall. And using simple shell-script it is possible to add or remove rules to this firewall. Thus during runtime we can control the IP-stack of the machine in a such way that some particular packets are blocked, and we can switch on/off this behavior.

It was decided to block the replies from the standby CA Gateway using firewall rules. Another option would be to block broadcast requests or replies to/from the CA servers, but as it was described in the previous section that would break normal operation of the CA Gateway. If the replies from CA servers are not seen by CA Gateway, then it assumes that that PV is not present on the subnet and it will mark it as such in its internal database. Therefore when the switchover will occur and client will try to search for that PV on that standby gateway it will not reply to them, because of that record. On the other hand we could block the request from the CA client all at once, then the database of the standby CA Gateway would be empty until the failover. But when the failover would happen and the clients start to connect through that gateway, a huge number of broadcasts would be required on the servers side. Therefore it is very advisable to keep a track of the PVs on the standby CA Gateway and block only the replies from it to the CA clients until it becomes master. In this scheme when the failover happens, the CA Gateway will be able to connect the CA without issuing additional CA broadcast storm on the servers subnet.

The required shell scripts that block and unblock the replies from the CA gateway were written. Another script was written to monitor the status of the CA Gateway. Unix “kill” command was used for that. When this job was done it was time to implement the RMT driver, that would call these scripts.

4.2.2 General Purpose External Command Driver for RMT

When I had to implement the driver, that would run the scripts, described above, I chose to do it in some generic way, so that driver could be used for other purposes. This driver uses simple text configuration file, where user can specify any arbitrary external program to be called when “Start”, “Stop” and “Check status” functions from RMT are called. It can also easily be extended to support other function of the RMT API. This allows to use RMT run any external command on some particular event. It gives very rich capabilities to monitor and log the work of RMT and redundant software implemented with its help. For example every time the switchover happens, RMT can send an e-mail or play a sound or do whatever else, or use the status of some program running as one of the aspects in decision making “switch-over” or “not switch-over”.

4.2.3 Redundant Channel Access Gateway Implementation summary

The redundancy without load-balancing was implemented by using the RMT as a stand-alone application, which runs separately from the CA gateway. It provides the benefit of not modifying the source of the gateway, but raises the problem of how to control the gateway. Because the RMT and the CA Gateway are separate processes, RMT cannot start and stop CA Gateway using RMT-API.

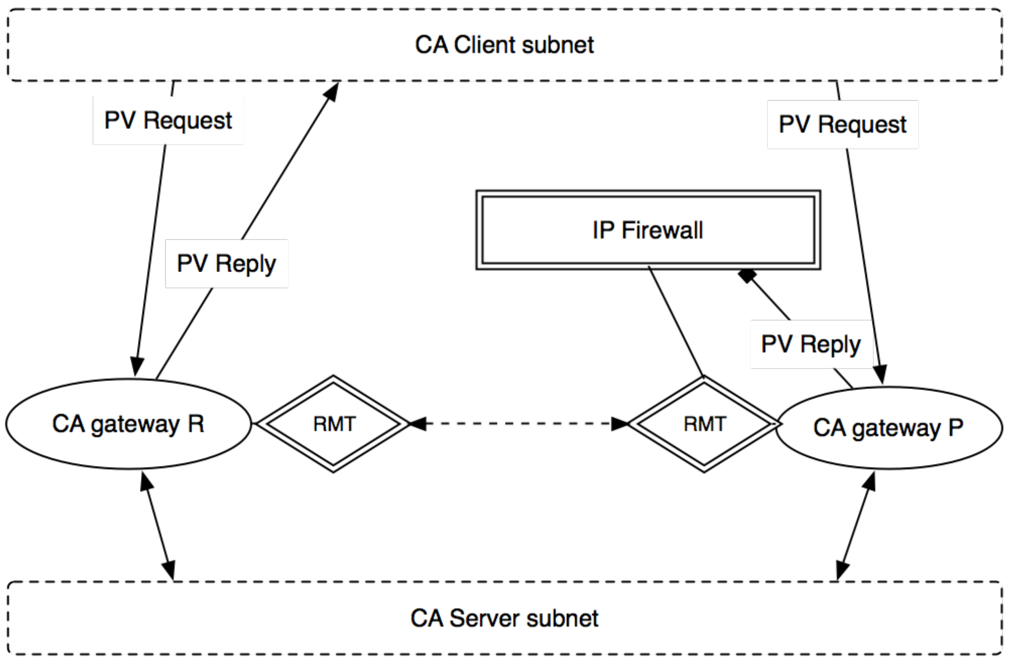

Figure 4.6: PV reply is blocked by RMT on the SLAVE CA Gateway

Therefore, the chosen solution was to use firewall rules to block replies from the Gateway process (Figure 4.6). In order to add and delete firewall rules, the RMT-driver was implemented that would run external scripts. Upon becoming a master this driver makes a call to an external "start-script" (which may be any executable file), and upon becoming a slave, it calls an external "stop-script". This script driver may be used in other applications and may call any external application on a specified status change. In our case, stop-script adds a firewall rule that blocks replies from the gateway to the clients, while start-script removes this firewall rules, making it possible to send replies to the clients. Hence, at any given time there is only one replying gateway that replies to the clients. The other gateway is on standby. This scheme worked well and hence I proceeded to the next step - implementing load-balancing CA gateway.

4.3 Redundancy with load-balancing

This task appeared to be more tricky. Before I decided to use RMT as a standalone application, in order to reduce impact on the original gateway. However, to introduce load-balancing the RMT needs to have more control over the gateway. At least a load-balancing CA gateway has to be informed of the current status of its partner. I our approach of having them as separate processes it was decided to use the Unix signals mechanism to inform the gateway process about the status change. Some modification was done to the gateway source, which allows the reception of two signals. One of them signifies that the partner has become “alive”, and the other signifies that the partner has become “dead”.

The logic of a load-balancing gateway is simple: when the partner is alive, every other reply from the gateway includes its partner's IP address (this is the same functionality of the CA which is used in the CA directory service).See Figure 4.7 and Figure 4.8. When the partner is dead, the Gateway replies normally.

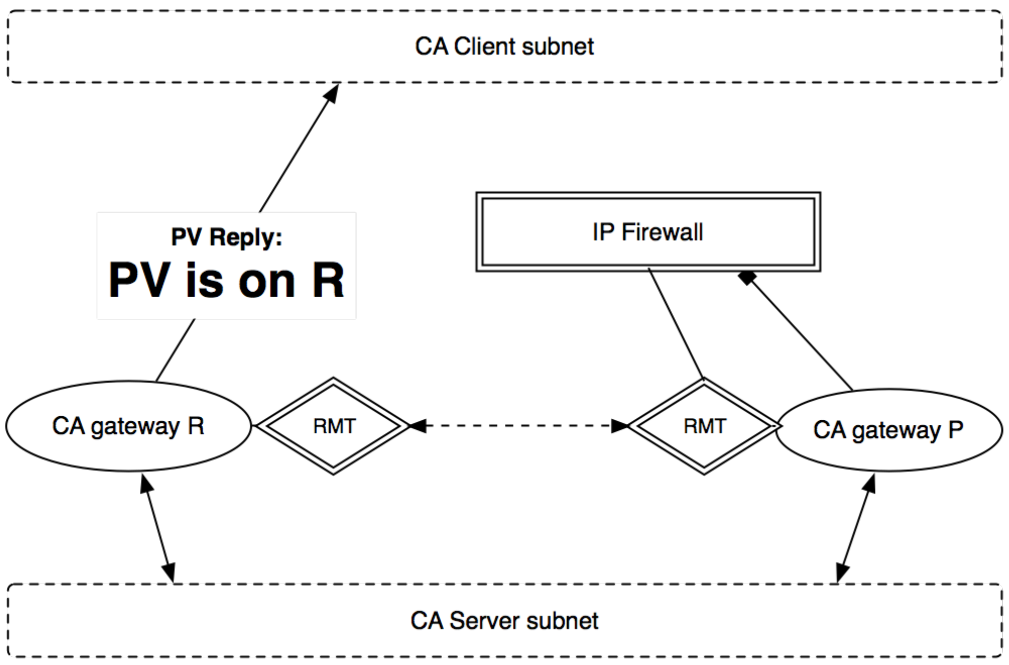

Figure 4.7: Load-balancing gateway – first reply

Hence, both the gateways perform the same function if they are alive at the same time. Upon receiving a PV search request from a CA client they perform a search on the IOC network; if the search is a success, they create a “virtual-circuit connection” to the corresponding IOC. At this stage, the CA gateway adds this PV to the list of known PVs. Subsequently, a reply to the client is sent (containing either the partner's IP address or it's own IP address). However, on the slave side, the RMT's "script-driver" adds a firewall rule blocking replies. Therefore, the clients receive only one reply from the master. The master load-balances the further requests between itself and its partner. Therefore, after receiving a reply from the master gateway, the CA client establishes a connection to one of the two gateways. If we would block the incoming request from the client instead of the reply from the gateway, the client cannot connect to the slave gateway later because its list of know PVs would be empty and it will deny all incoming connections.

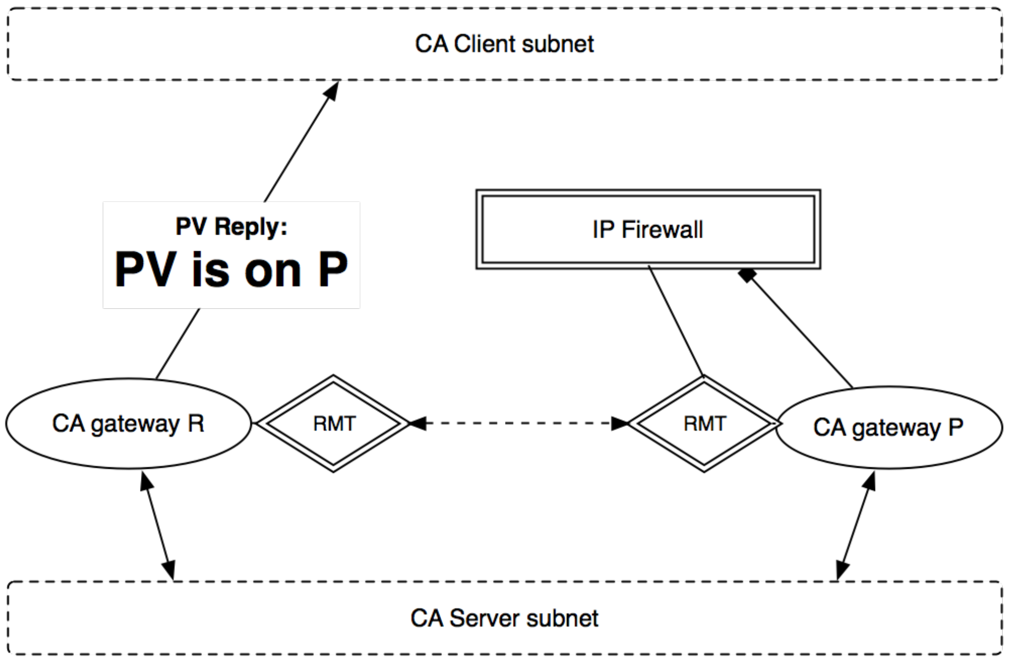

Figure 4.8: Load-balancing gateway – second reply

4.3.1 Conclusion

The load-balancing Channel Access gateway has several benefits compared with the non-load balanced redundant CA Gateway and original CA Gateway. The load balancing gateway provides all the benefits that redundant CA Gateway does plus more. It can handle two times more load and bandwidth than original CA Gateway and provides much more stable environment that original CA Gateway and even better than redundant CA gateway. In case of a failure or switchover only half of the clients would need to reconnect trough the standby gateway, because the other half is already connected through the standby. It means that for half of the clients it will not be even noticeable that one of the gateways went down. The other benefit of load-balancing is faster response, less load and better utilization of available resources. In case of a redundant only CA GAteway the standby machine is not performing any useful work other than waiting to catch up the operation in case of a failure. Load balancing gateway uses both machines and thus can handle twice as much connections in the extreme case, or just put less load on each of the machines.

As a result of putting all this together, we achieve a load-balancing redundant CA gateway. Some minor changes to the CA Gateway source code are required. These changes include signal handling, load-balancing functionality and new command line options for configuring the IP address of the partner and signal numbers. Altogether the load balancing Channel Access gateway provides better performance and stability along with redundancy.

5 Redundant IOC on Advanced Telecommunication Computing Architecture: Reliable software + Reliable hardware

ILC project chose the Advanced Telecommunication Computing Architecture platform for its Control System[1]. Other projects(DESY/XFEL, PANDA, ITER, etc.) show growing interest to this standard as well. Although ATCA is already a mature hardware platform, providing extensive capabilities (including redundancy), the accelerator control software for this platform is yet to be developed. The telecommunication industry makes a great effort in developing high availability standards and middleware, that can be used by application developers. In this work I present my experience using these standards and middleware to add a support for ATCA hardware to the Redundant IOC.

5.1 Advanced Telecom Computing Architecture (ATCA)

ATCA standard is defined by PCI Industrial Computer Manufactures Group with 100+ companies participating. The first standard was published in 2004 and it has been already widely supported by major vendors. It is primarily targeted to requirements of carrier grade communications equipment and incorporates the latest trends in high speed interconnect technologies, next generation processors and improved reliability, manageability and serviceability. Availability of ATCA crate is designed to be 99.999%. That and other features of ATCA standard made it the platform of choice for the ILC control system.

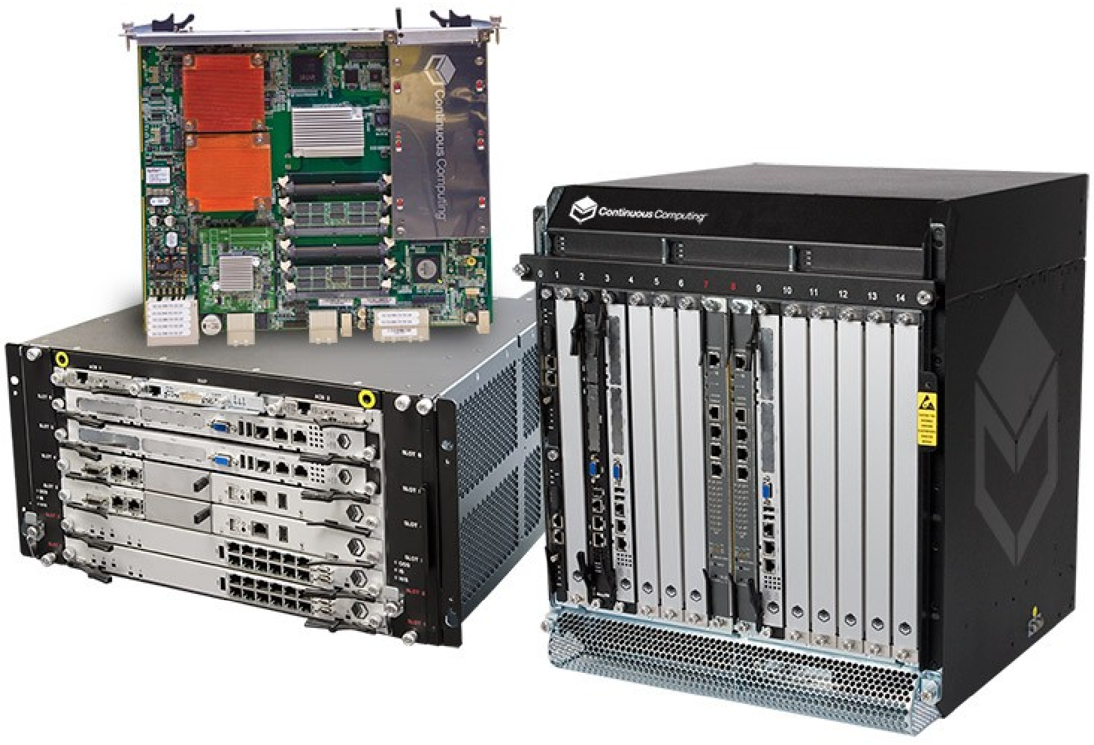

Figure 5.1: ATCA crates and boards

ATCA defines extensive monitoring and controlling capabilities. All the components of an ATCA system are interconnected via (usually) redundant Intelligent Platform Management Bus (IPMB).

One particular board – Shelf Manager (SM) - plays a centre role in the management of the system(Figure 5.2). SM is responsible for polling and controlling other hardware via IPMB. Usually SM implements some simple logic for monitoring overall system health and some fail-over procedures (which are vendor and hardware specific).

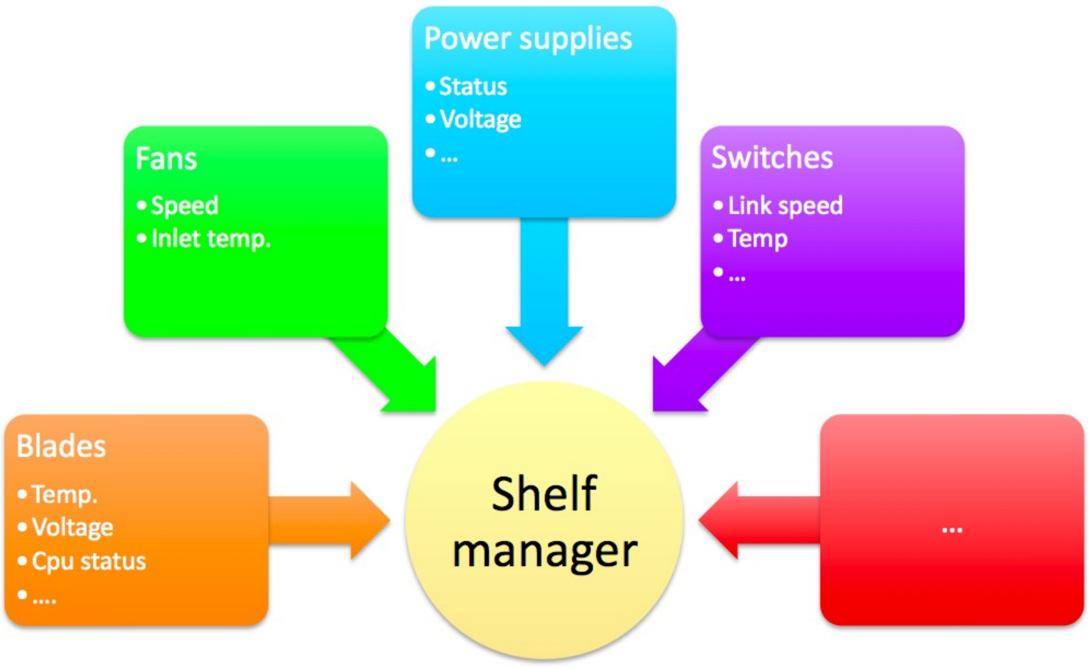

Figure 5.2: Shelf manager - local control center within the ATCA crate

SM provides the access to the hardware for third-party via Hardware Platform Independent library (HPI) or SNMP. HPI covers all the differences in actual implementation of the ATCA standard and provides hierarchal representation of all available hardware.

5.2 SAF Specifications

5.2.1 Service Availability forum

The Service Availability Forum™ is a consortium of industry-leading communications and computing companies working together to develop and publish high availability and management software interface specifications. The SA Forum then promotes and facilitates specification adoption by the industry. Two main specifications by SA Forum are - Hardware Platform Interface (HPI) and Application Interface Specification (AIS)[8]. The SAF specifications are primarily aimed at telecom industry, but they can be used for physics HA applications as well. SAF specifications represent current best practices in telecom industry and guidelines for building HA systems. Even if the accelerator society decides not to use them, its beneficial to get acquainted with these specifications.

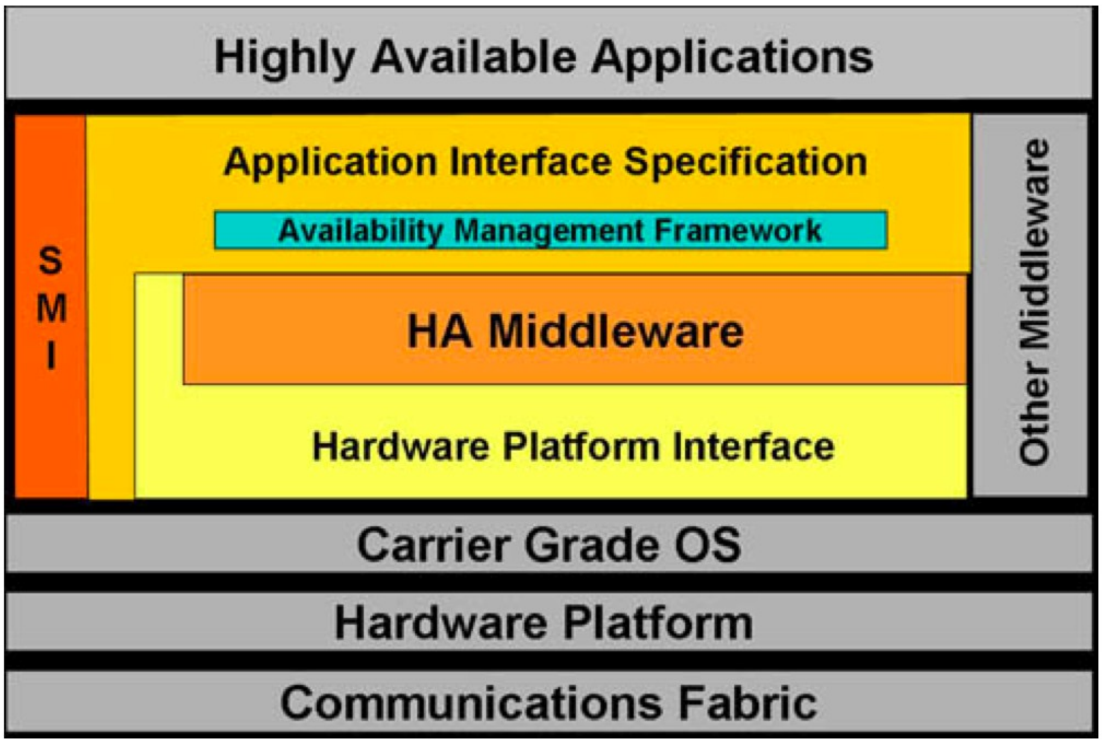
Highly Available Applications
S M I
Application Interface Specification
Availability Management Framework
HA Middleware
Hardware Platform Interface
Other Middleware
Carrier Grade OS
Hardware Platform
Communications Fabric

Figure 5.3: SAF Application Architecture

The SAF HA application architecture can be seen on the Figure 5.3. Where SMI stands for Software Management Interface - part of the AIS specification. Basically HPI provides hardware independent API to address the hardware and AIS provides the API and infrastructure needed for HA, such as synchronization, event propagation etc.

5.2.2 Hardware Platform Interface (HPI)

The HPI specification separates the hardware from management middleware and makes each independent of the other. The HPI concerns, primarily, the management of individual hardware components, called entities. The main purpose of the HPI is hardware control and monitoring. The HPI provides hot swap, the ability to replace hardware components within an operational system. Many of the mechanisms for providing service availability (such as managing standby components, failover and fault recovery) are provided by the AIS, rather than by the HPI. However, the HPI is self contained and can operate independently of the AIS.

The HPI-to-AdvancedTCA Mapping Specification details how the SA Forum's HPI maps to PICMG's AdvancedTCA specification. By standardizing how the two specs should be implemented together, the mapping specification gives developers a standard method to access functionality in both specifications, thereby saving time, money and resources. Benefits of using HPI :

- Shorter development cycles
- Development cost savings
- Lower total cost of ownership
- Improved design flexibility
- Reduced development risk
- Faster innovation

The HPI is primarily used with ATCA, but HPI is a platform independent specification, therefore it does not require ATCA. There are HPI implementations that can run on common server-grade IBM PC compatible computers (OpenHPI)[14].

5.2.3 Application Interface Specification (AIS)

The SAF AIS standardizes the interface between SAF compliant High Availability (HA) middleware and service applications. The core element of the AIS is Application Management Framework (AMF). AMF provides service availability by coordinating redundant resources within a cluster to deliver a system with no single point of failure. Another important part is Software Management Framework (SMF) embodies standard mechanisms to deploy, configure and monitor the software used within the cluster. Besides AMF and SMF, the AIS defines a set of auxiliary services, that altogether provide standard API and infrastructure for building HA applications.

Although, in this research the AIS specifications were not used, they are should be a matter of a great interest for anyone involved with HA applications development for the ATCA platform.

5.3 Redundant IOC on ATCA

As mentioned earlier, the ATCA platform requires a special software to be developed in order to utilize its capabilities. One of the features of the ATCA platform is the implied redundancy of the major components of the system. This makes it a perfect candidate to use in conjunction with the EPICS redundant IOC.

It is possible to run “vanilla” RIOC on ATCA hardware. It is beneficial to use ATCA to run EPICS RIOC in a sense of using reliable hardware, but it does not differ much from using two separate PCs. ATCA provides extensive monitoring and management capabilities, which are not utilized by “plain” EPICS RIOC. HPI driver for EPICS RIOC was developed in order to make EPICS RIOC aware of ATCA hardware.

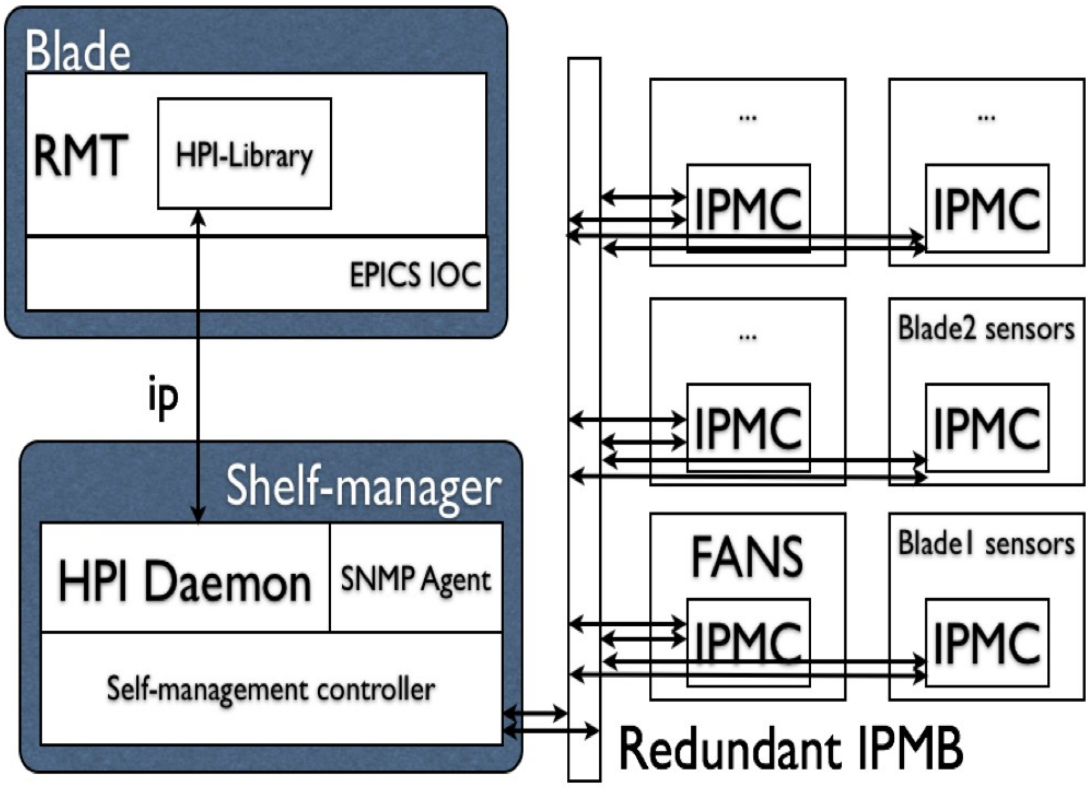

Figure 5.4: Redundant IOC on ATCA architecture

Within the ATCA shelf all the components are interconnected via redundant Intelligent Platform Management Bus (IPMB) as shown on the Figure 5.4. Instead of accessing IPMB directly, we used HPI library to develop an extension for EPICS RIOC (in a form of RMT driver). That extension allowed to include the status of the ATCA hardware into the fail-over decision process. Due to high level of abstraction and hardware independence of HPI it is possible to monitor any available set of sensor on the system without modification to the source code. The configuration is done by editing plain text configuration file.

The ability to monitor the hardware of the system allowed us to improve the reliability of EPICS RIOC. For example, if a CPU temperature starts to rise, there is some limited time before it will crash. And if properly monitored, we can initiate the fail-over process before the actual hardware failure happens. For the EPICS RIOC applications it gives us two major benefits:

- Fail-over happens while the system is still working, so actual transition happens in a stable and controlled environment.
- Channel Access connections can be gracefully closed. That will drastically reduce the reconnect time for Channel Access Clients (CAC). Normally in case of “hard” failure of a master RIOC it takes up to 30 seconds for CAC to reconnect to the slave RIOC (30 seconds is a default EPICS connection time-out, actual time-out may be changed by user). Even though the slave EPICS RIOC notices the problem instantly and within 2 seconds takes over.

The usage of HPI allowed us to avoid any hardware specific programming in the first place, but also made our solution portable to platforms other than ATCA. For example an open source implementation of HPI - OpenHPI - it can run on top of Linux 2.6 sysfs. And if the hardware allows provide access to the sensors. Therefore, without any source-code modification our “hardware-aware” EPICS RIOC can be used on server-grade Linux computers.

6 EPICS test system automation

As it was described in second chapter, in terms of achieving reliability software is principally different from hardware. Software failures cannot be fixed by replacing the failing software component with the new one. To eliminate the occurrence of the same failure in the future the software has to be analyzed and rewritten. Naturally all software has “hidden” failures, that will occur only when certain conditions are met. Until then the software keeps running as supposed and there is no sign that someday, when these conditions happen, the software will fail. Therefore thorough testing and quality control is such an important thing in software development. Testing of the software is a continuos process, ideally it happens along with the development of the software. There are several “levels” of software tests, depending on the target of the test:

- Unit testing: is aimed to test the smallest piece of code, such as function or class. Unit testing ensures that these small pieces perform according to the specification. EPICS software is distributed with unit test system included, anyone can use it by issuing the command “*make runtests”.*
- Integration testing. The goal of integration testing is to ensure that the blocks of code tested at the unit test stage are working properly when combined together.
- System testing is performed on a complete system, when all the building blocks are connected together. System testing ensures that the system as a whole performs correctly and according to the specification.
- System integration testing is aimed to verify system's interaction with other third party components.

6.1 EPICS Testing

EPICS is a complicated and large software project and of course it has its quality and test systems. EPICS has extensive unit-test system, developed within the years of EPICS existence. These unit-test are supplied with every version of EPICS distribution and could be executed after building the base using command “make runtests”. It is advised to run these test on every machine where EPICS is used to ensure compatibility. EPICS supports multiple Operating

Systems since release 3.14 when EPICS libOSI (Operating System Independent Library) was introduced. Both client and server applications can run on Linux, Windows, Mac OS X, Solaris, vxWorks, RTEMS, Tru64UNIX, FreeBSD.

It is very common situation when even within one laboratory more than one OS is used in EPICS environment. vxWorks and RTEMS operating systems are used for real-time applications, software IOC running on Linux and Windows are used for less critical applications. Client programs such as display managers and archivers are usually run on different flavors of unix-like systems (Linux, Mac OS X) and Windows. Versions of these OS also may differ very widely. And from version to version, or from one distribution to another there may be so many significant and not so much differences. Besides OS differences, EPICS can run on a wide variety of hardware platforms. Thus the variety of software and hardware platform that EPICS can be used on is really huge. All this makes it virtually impossible to test all the possible combinations of operating systems and hardware. During the release phase EPICS distribution is tested by the core-team, but this test does not cover all the usage cases of EPICS in the real world. Therefore it is essential for the end users of EPICS to test it on the usage-sites themselves. As mentioned earlier unit-testing is pretty simple and user-friendly process, but everything that goes beyond unit-testing is not that easy in such a heterogeneous environment where EPICS is usually used.

6.1.1 MrkSoftTest package

Anyway, for years there has been a package called mrkSoftTest, which represents a system integration level test package for EPICS. It begins its history when EPICS only supported vxWorks, but even then it could have been used on different hardware platforms. Differences such as byte-order have to be checked to ensure system interoperability. Later when EPICS became multi-platform the necessity of these checks increased even more. So nowadays this package includes several tests, which check CA network links, alarm functionality, correctness of conversion functionality between local and remote systems. Typical test scenario involves one or more IOCs and several Channel Access clients.

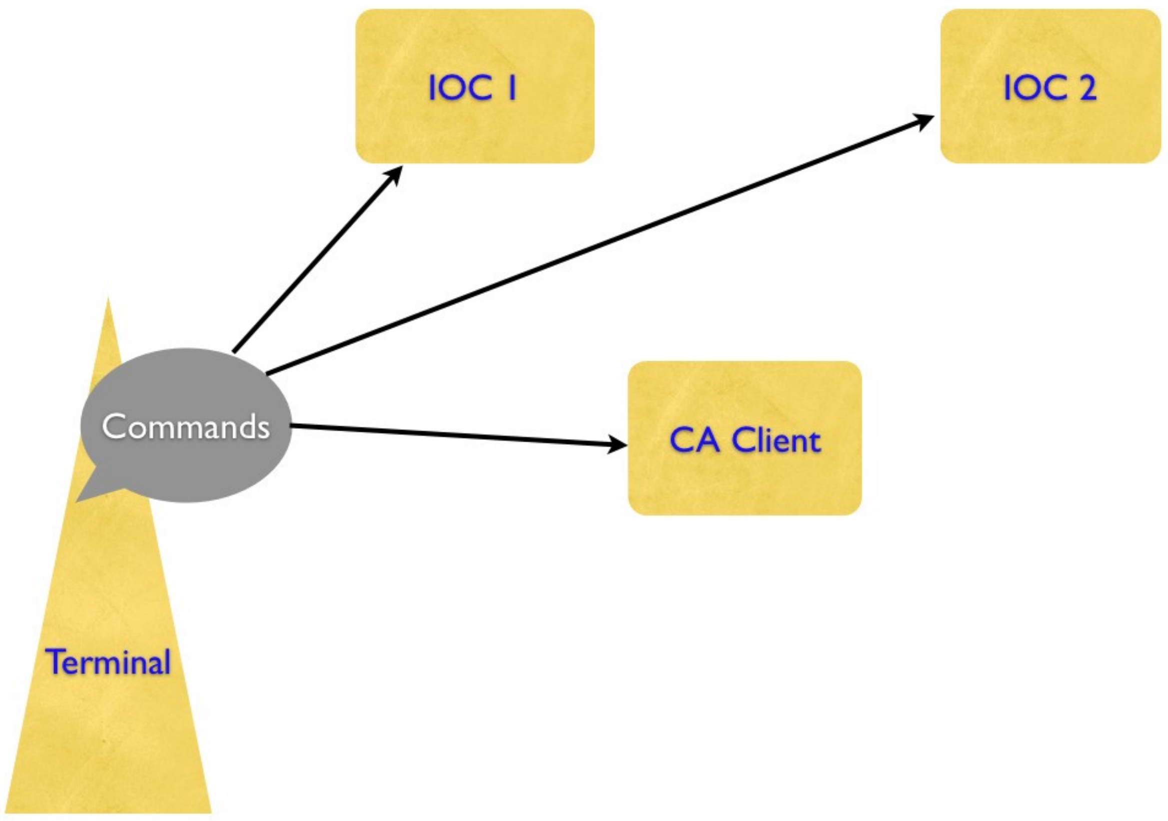

Figure 6.1: Typical test scenario – start the IOCs and CA clients manually

6.1.1.1 Typical Test Scenario

Common test scenario can be described by the following actions (steps), including the operator actions (bold font):

- **Connect to remote station via SSH/Telnet/ SH.** Figure 6.1
- **Configure & start EPICS IOC.** Figure 6.1
- **Repeat steps 1-2 for required number of times to start other IOCs Start CA client(s), locally or remotely Issue the required sequence of control commands for IOC and/or start additional CA clients.** Figure 6.1
- IOCs and CA clients interact, running the test software. Figure 6.2

- IOCs and CA clients print out the results. Figure 6.2
- **Gather the output from all the programs.** Figure 6.3
- **Compare the output to the reference file.** Figure 6.3
- **Shutdown all the IOCs and CA clients.** Figure 6.3

This procedure is quite complex by itself, but becomes more complicated when we introduce differences between OS and Hardware. Commands and output may differ for different architectures and particular configuration.

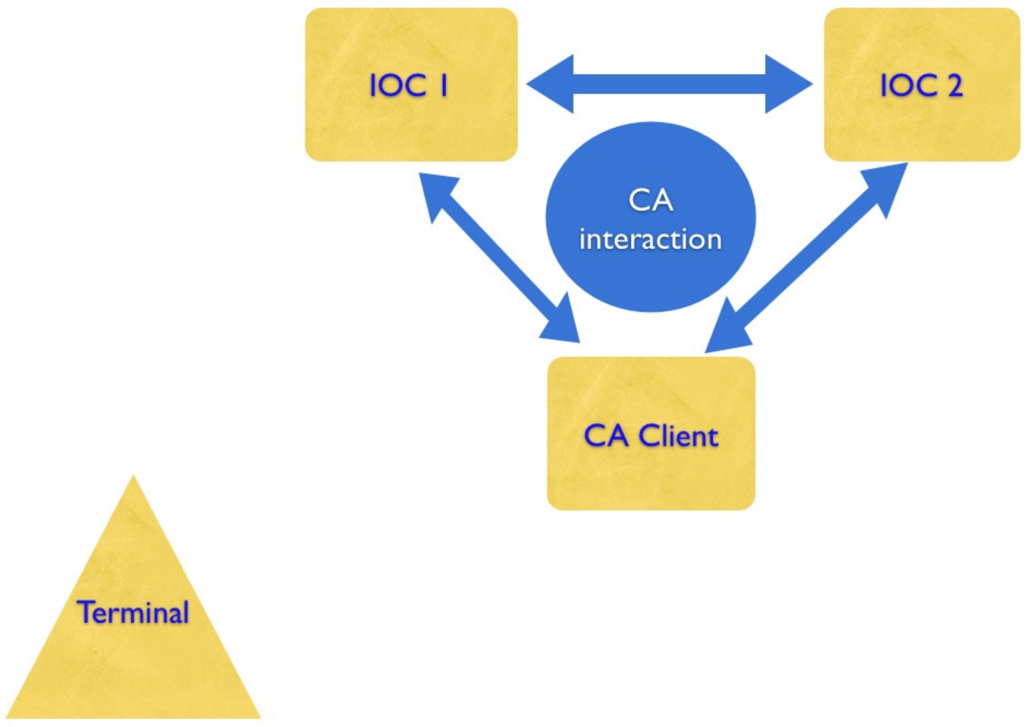

Figure 6.2: Typical test scenario consists from several interacting components

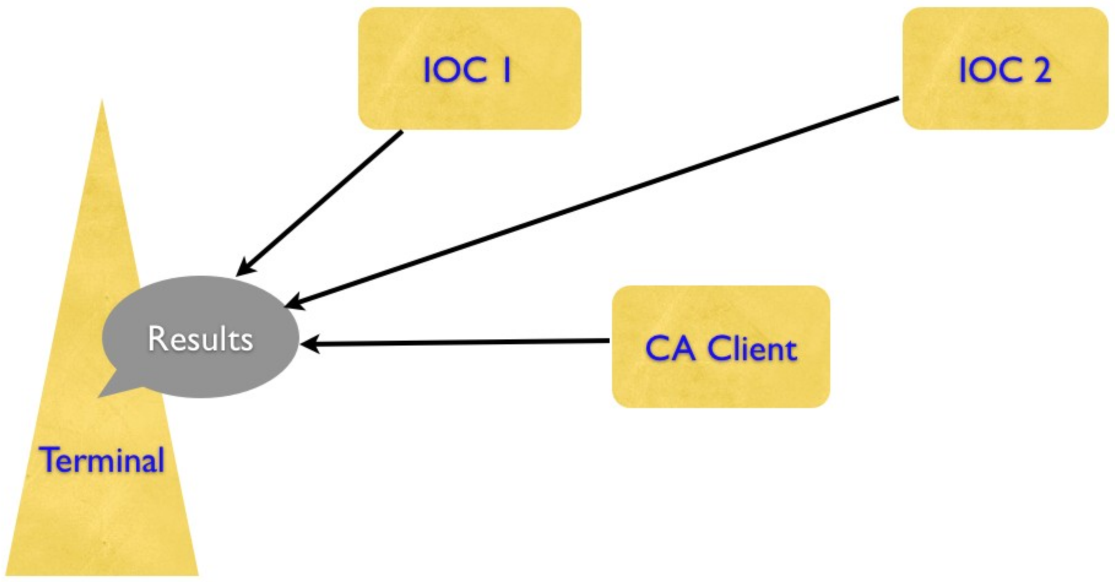

Figure 6.3: Typical test scenario – gather and analyze results

As it can be seen, human interaction takes very big part of these system integration tests. But most people do not remember how to run these test, and have to read the instruction every time they run these test. Overall this becomes very time and effort consuming operation. Because of the difficulty and inconveniences of mrkSoftTest package only a few people do run these tests, mostly they are EPICS-core developers. General users do not run these test at all, even though it is highly advisable. What could be done to make people use these test more often? The answer is simple: make these tests easy to run! Something like “*make runtests*”.

6.2 Solution: Automation

To solve this problem EPICS test automation package (EPICS TAP) was developed. EPICS TAP simplifies the process of running system integration tests and hides all the low level details from the person running these tests. EPICS TAP was developed using high level scripting language Ruby. EPICS TAP framework uses a common to unit testing terminology and it was partly inspired by my exposure to Ruby's Test::Unit framework. Although, the interface might look familiar, the implementation is very different. The main purpose of the EPICS TAP framework is to executes a series of external commands on a set of network connected computers, and to hide the possible differences from the end user.

6.2.1 EPICS Test Automation Package Architecture

EPICS TAP provides an object oriented environment for developing system tests. EPICS TAP consists from several classes, that are aimed to encapsulate the differences and details of test scenarios from the end-users and to simplify the implementation for developers. All classes are defined in the EPICSTestUtils and the Cfg modules. The class diagram is shown on the Figure 6.4.

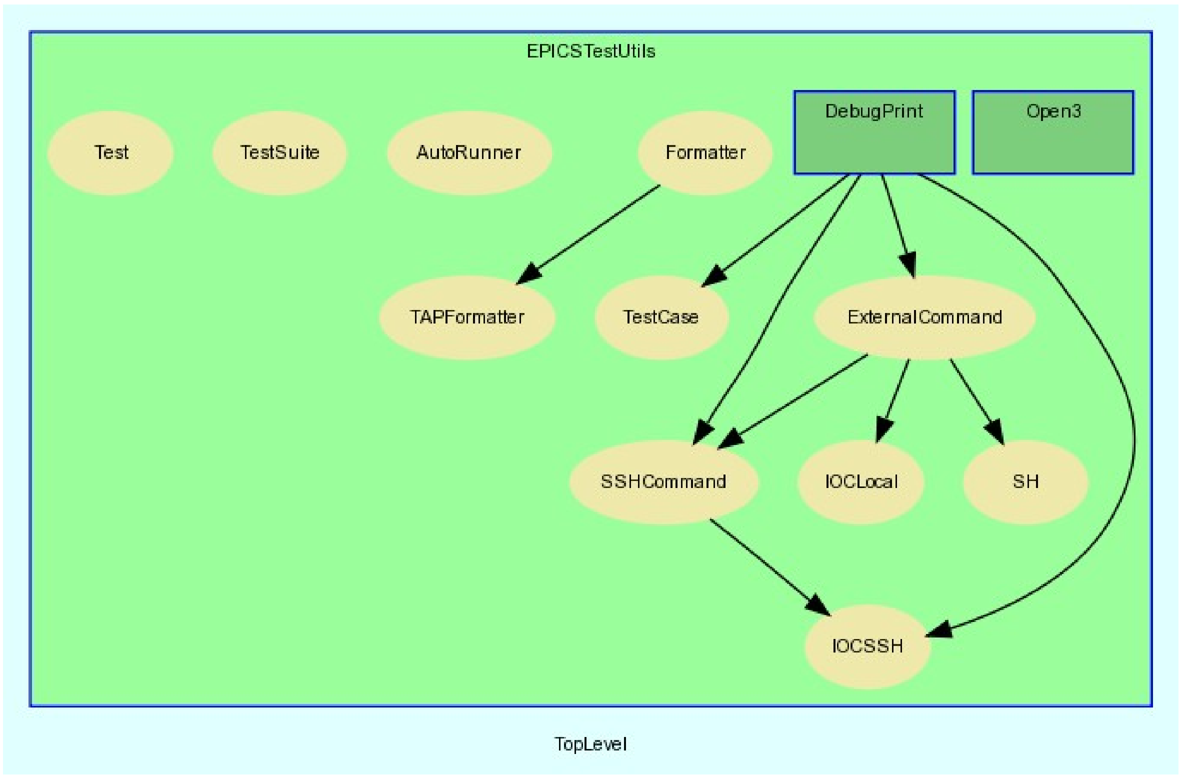

Figure 6.4: EPICS TAP class diagram

6.2.1.1 Cfg module

This module defines some functions and a class to work with EPICS TAP configuration file. This system is designed to be very flexible and to allow easy to create/understand hierarchal configuration file.

class Cfg::Config < Hash

This class extends the Hash class. It redefines the default *method_missing* method, so it searches itself for the required key and if not found, then the parent hash is searched for the required key. It allows to use '.' to access the values in the configuration and use hierarchal definitions of values.

def Cfg.default

default configuration parameters. The configuration file must contain *:defaults* section

def Cfg.h

shortcut for hosts configuration parameters defined in the *:hosts* section of the configuration file.

def Cfg.load

This is the core of this module. It read the configuration file and builds the configuration tree, that can be used in EPICSTestUtils.

Here is a simple example of Cfg capabilities, such as path-like naming, hierarchal structure and different methods to access the data.

```
require 'EPICSTestUtils'

Cfg.load("config.yml")
Cfg.h.each_pair do |k,v|
  puts k + "=" + v.hostname
end
puts Cfg.h.localhost.hostname
puts Cfg.h["localhost"]["hostname"]
puts Cfg.c.TestCA.IOC0.hostname
puts Cfg.c.TestCA.COMMAND0.hostname
puts Cfg.c.TestAlarm.COMMAND0.hostname
puts Cfg.c.TestAlarm.IOC0.hostname

puts Cfg.h.localhost.hostname
puts Cfg.h.localhost.default
```

and the corresponding configuration file in YAML format:

```
# = test configuration file
# good for reference
:default: &default
  debug_level: 4
  user: tyoma
  topDir: /Users/tyoma/epics/soft-test
  epicsTopDir: /Users/tyoma/epics/base-3.14.10
  binDir: bin
  referenceDir: reference
  autostart: yes

# == :hosts:
# should contain at least +localhost:+ definition, as long as others which you intend
to use for your tests
# typical config has +localhost host1 host2 host3+
:hosts: &hosts
  localhost:
    hostname: tyomac
    epicsHostArch: darwin-x86
  host1:
    epicsHostArch: freebsd-x86
    hostname: burdock
    epicsTopDir: /usr/home/tyoma/base-3.14.8.2
    topDir: /usr/home/tyoma/soft-test
  host2:
    epicsHostArch: linux-x86
    hostname: durian
    epicsTopDir: /usr/users/control/epics/R3.14.9/base
    topDir: /usr/users/tyoma/epics/soft-test

# So... the rigth way to configure all this stuff should be.
# 1. Default config < Host Config
#
# "<<" Means include the named variable

:defaultIOC: &defaultIOC
  ioc: common

:defaultCommand: &defaultCommand
  type: SH

:defaultCommandSSH: &defaultCommandSSH
  type: SSH
  host: localhost
  user: tyoma

:defaultTestCase: &defaultTestCase
   IOC0:
    <<: *defaultIOC
   COMMAND0:
     <<: *defaultCommand
   COMMAND1:
     <<: *defaultCommand

:testCases:
  TestAlarm:
   IOC0:
    <<: *defaultIOC
    bootDir: iocBoot/iocalarm
    cmd: stcmd.host
    host: host2
   COMMAND1:
     <<: *defaultCommandSSH
   COMMAND0:
     <<: *defaultCommand

  TestCA:
    <<: *defaultTestCase
    IOC0:
      <<: *defaultIOC
      bootDir: iocBoot/iocca
      cmd: stcmd.host
      host: localhost
      autostart: no
    COMMAND0:
      <<: *defaultCommand
      autostart: no
    COMMAND1:
      <<: *defaultCommand
      autostart: no

  TestConvert:
    <<: *defaultTestCase
    IOC0:
```

```
    <<: *defaultIOC
    ioc: convert
    bootDir: iocBoot/iocconvert
    cmd: master.main
    host: localhost
  IOC1:
    <<: *defaultIOC
    ioc: convert
    bootDir: iocBoot/iocconvert
    cmd: client.main
    host: host1
  COMMAND0:
    <<: *defaultCommand

TestLinkInfoLocal:
  IOC0:
    <<: *defaultIOC
    ioc: linkinfo
    bootDir: iocBoot/ioclinkinfo
    cmd: stcmdlocal.host
    host: localhost

TestLinkInfoRemote:
  IOC1:
    <<: *defaultIOC
    ioc: linkinfo
    bootDir: iocBoot/ioclinkinfo
    cmd: stcmdremote.host
    host: host2
  IOC0:
    <<: *defaultIOC
    ioc: linkinfo
    bootDir: iocBoot/ioclinkinfo
    cmd: stcmdlocal.host
    host: localhost

TestPut:
  IOC0:
    <<: *defaultIOC
    bootDir: iocBoot/iocput
    cmd: put.main
    host: host2

TestSoftCallback:
  IOC0:
    <<: *defaultIOC
     bootDir: iocBoot/iocsoftcallback
    cmd: stcmd.host
    host: host2
  COMMAND0:
    host: localhost


TestRemoteIOCSh:
  IOC0:
    <<: *defaultIOC
    bootDir: iocBoot/iocalarm
    cmd: stcmd.host
    host: host2
TestCase2:
 <<: *includes
 ioc: exampleIOC
 varname: val
 tmpDir: tmp/exioc
 IOC0:
  <<: *defaultIOCSSH
  host: host1
  bootDir: iocBoot/iocalarm
  cmd: stcmd.host
 COMMAND0:
  host: host1
 COMMAND1:
  host: host2
```

6.2.1.2 EPICSTestUtilModule

This module defines the classes used to build test scenarios and several helper classes and methods to simplify the process.

AutoRunner

This is a helper class, that allows to run the same exact TestCases within a TestSuite or standalone. If there is no TestSuite instance defined, then AutoRunner instance creates an instance of TestSuite and add there all the TestCases visible in the current runtime scope. *EPICSTestUtils.run* variable defines whether AutoRunner is executed or no.

TestSuite

Test Suite is a collection of Test Cases. Test Cases are independent instances described in the next section. Test Suite is an administrative instance can be used to group test in any possible way. When the TestSuite instance is created, by default it sets the *EPICSTestUtils.run = false* variable to disable the AutoRunner. Test Suite class provides the following hooks available for developer:

- suite_setup - empty by default
- suite_teardown - empty by default
- run - runs all the TestCases within the TestSuite.

Example of the TestSuite usage:

```
testSuite << TestAlarm.new(formatter,config)
testSuite << TestCA.new(formatter,config)
testSuite << TestLinkInfoLocal.new(formatter,config)
testSuite << TestLinkInfoRemote.new(formatter,config)
testSuite << TestPut.new(formatter,config)
testSuite << TestSoftCallback.new(formatter,config)
testSuite << TestConvert.new(formatter,config)
testSuite.run
```

Test Case

Test Case represents a set of tests that share the same environment, configuration and can be run consequently. For example if we want to test channel access put, get and monitor functionality on two IOCs, then we can put these three tests into one test case. Each test within *TestCase* is implemented as a member function. The *TestCase* class defines many hooks available for developers to redefine:

- *setup_iocs* - this function finds IOCs configured for this particular test case. The default implementation reads the configuration file and configures the IOCs according to it. Each IOC is represented by an object of *IOCLocal* or *IOCSSH* class, local and remote consequently.
- *setup_commands* - same as previous, but for arbitrary external commands, that may be needed during test. External Commands are represented by *SH* or *SSHCommand* classes. (Channel Access Clients usually are these external commands)
- *global_setup* - default is empty. This is a placeholder function available for test developers. It is called before anything is executed, but IOCs and commands are already configured and corresponding objects are available.
- *start_iocs* - starts the IOCs
- *start_commands* - start the commands
- *global_setup_after_start* - empty by default, at this point IOCS and commands are usually running
- *setup* - empty by default, called before each test in a test case
- *teardown* - empty by default, called after each test in a test case
- *global_teardown* - empty by default, called after all tests in a test case
- *stop_iocs* - stops the IOCs

- *stop_commands* - stop the commands

Some of these functions are just placeholders and some have some default behavior, in any case they represent very flexible scheme to accommodate a wide variety of possible test scenarios. And if it is not enough it can be easily extended by redefining the *case_setup* and *case_teardown* functions.

Actual tests are implemented either as class member functions with the name starting with *test_* or as objects of *Test* class (this is deprecated interface). The *TestCase* default implementation automatically finds the methods starting with *test_* and executes them in alphabetical order.

Here is the example of test case:

```
#!/usr/bin/env ruby
require "EPICSTestUtils"
class TestCA < EPICSTestUtils::TestCase
  def global_setup
    cmd[0].startcmd="#{cmd[0].localBinDir}/testcaput 5000 5"
    cmd[1].startcmd="#{cmd[1].localBinDir}/testcaget 5000 1"
    @client_caput = cmd[0]
    @client_caget = cmd[1]
    @unresponsive_message = /Warning: "Virtual circuit unresponsive"/
    @disconnect_message = /Warning: "Virtual circuit disconnect"/
  end

  def setup
    cmd[0].start
    cmd[1].start
    ioc[0].start
  end

  def teardown
    cmd[0].exit
    cmd[1].exit
  end

  def global_teardown
    ioc[0].exit
  end

  def test_courtesy_exit
    start_time = Time.now
    ioc[0].command("exit")
    response_caput = @client_caput.read_stderr(0.3)
    response_caget = @client_caget.read_stderr(0.3)
    assert(response_caput =~ @disconnect_message)
    assert(response_caget =~ @disconnect_message)
  end

  def test_sudden_death
    ioc[0].suspend
    response_caput = @client_caput.read_stderr(15)
    response_caget = @client_caget.read_stderr(15)
    assert(!(response_caput =~ @unresponsive_message or response_caget =~
@unresponsive_message))
    sleep 6
    response_caput = @client_caput.read_stderr(2)
    response_caget = @client_caget.read_stderr(1)
    assert(response_caput =~ @unresponsive_message)
    assert(response_caget =~ @unresponsive_message)
  end
end
```

Formatter Class

This is a generic class, that is used inside the *TestCase* to format the output of the tests. Actual formatters have to inherit from that class or define the same interface. *EPICSTestUtils* provides two predefined formatters: *TAPFormatter* - to format output in Test Anything Protocol and simple text formatter (*Formatter* class). Here is the *TAPFormatter* class:

```
class TAPFormatter < Formatter
  def header(context)
    puts "# #{context.title} starting"
    puts "1..#{context.count_test}"
  end

  def print_test_result(name, result, num = nil)
    if num
      puts "#{num} #{result} # #{name}"
    else
      puts "#{result} # #{name}"
    end
  end

  def report(context)
    if context.result == :OK then puts "ok - #{context.title} #{context.description} #
#{context.explanation}"
    else puts "not_ok - #{context.title} #{context.description} # #{context.result}
#{context.explanation}"
    end
  end
  def put(string)
    puts "# #{string}"
  end
  def footer(context)
    puts "# #{context.title} is over"
  end
end
```

6.2.2 Conclusion

Introduction of EPICS TAP made running tests as simple as typing *“./runAlltests.rb”* and then wait until it finishes. Depending on the configuration some changes are required to the configuration file (such as ip-addresses and OS system types). EPICS TAP ruby classes can be used to write additional test cases as well. EPICS TAP provides fully automated infrastructure for testing EPICS implementations.

Compared to manual testing, an automated testing saves a lot of human time and prevents the chance of a human mistake, recalling the list from section 6.1.1.1, now the sequence of actions looks like this:

- **Edit the configuration file**
- **./runAllTests.rb**
- **wait for the tests to finish (i.e. Go drink coffee)**
- **Get the result in simple readable format:**

```
# TestAlarm starting
1..1
1 OK # test_alarms
# TestAlarm is over
# TestCA starting
1..2
1 NOT_OK # test_courtesy_exit
2 OK # test_sudden_death
# TestCA is over
```

7 Conclusion and summary of the results

In this study, improvements to reliability accelerator control system were performed. The results of this study are summarized and concluded as follows.

As it was shown in the Introduction and the second chapter of this work, availability requirements raise a new challenges for designer and implementors of the accelerator control systems. In this work a control system is viewed to be composed four basic components – hardware, software, people and procedures. Each of these components requires specific methods to achieve high availability. General guidelines to achieve high availability were presented in the second chapter. The rest of this work is devoted to improvements of hardware and software components of an accelerator controls system.

The key technique to achieve high availability for hardware is redundancy. Redundant systems allow to reduce the time needed to recover from a failure to mere seconds, compared to hours or days in a non redundant system (this is the time needed to replace or repair the failing component). In this work an implementation of EPICS redundant IOC is examined.

The original EPICS control software distribution lacked a support for redundancy. Therefore in collaboration with DESY EPICS redundant IOC was developed. I took part in this project on the stage of testing and operating system generalization to bring the EPICS redundant IOC to a wide range of operating systems. This was done using EPICS operating system independent library (libCom/OSI). Besides getting a multi-platform version of the EPICS redundant IOC, this work resulted in inclusion of redundant IOC support into the official EPICS distribution. The generalized version of the EPICS redundant IOC can be used on Linux, vxWorks, Mac OS X and other EPICS supported platforms. Part of this software, namely Redundancy Monitoring Task, was distributed as an independent library, that was used to add redundancy to other software.

Redundancy idea can be expressed, as elimination of a "single point of failure". High Availability design requires to reduce the number of such points to minimum. In recent EPICS control systems installations Channel Access Gateway software plays an important role. It is used for administrative and security management: it can be used to add a security layer, to divide a network into several segments in order to reduce the amount of broadcast traffic in each segment, or to reduce the load on the IOC by multiplexing the client connections. Naturally it becomes a single point of failure in such systems. Failure of a single caGateway may “cut-off” the whole network segment. In order to address this issue, a Redundant Channel Access gateway was developed as a part of this research. Further improvement of the Channel Access Gateway resulted in a load-balancing redundant Channel Access Gateway. Usage of load-balancing redundant gateway not only improves availability, but also provides better throughput and response time.

Recently high-energy physics community shows growing interest in a new hardware platform - Advanced Telecommunication Computing Architecture. The ILC project selected this platform for its control system. ATCA is a feature rich hardware platform, specifically designed for highly available applications. It provides extensive control and monitoring capabilities. In order to fully exploit these features new software must be developed. The design of ATCA platform implies the usage of redundant systems. For that reason I extended the EPICS Redundant IOC to support this platform. The developed ATCA driver allowed to monitor the status of all available hardware within an ATCA crate and use this information for better fail-over decision making. By constant monitoring of temperature, voltage, fan speed and other parameters it is possible to predict the failure and to switch to the stand-by controller before the main controller fails. This gives better systems stability and improved client performance. Reconnection times from clients are reduced to 2 seconds, compared to 30 seconds for non-ATCA aware redundant IOC. It brings improved availability to EPICS redundant IOC running on ATCA platform.

From the software point of view, high availability goes in parallel with High Quality of software. Therefore decent Quality Assurance methodology and techniques much be in place in order to achieve High Availability for the accelerator control system. To address this issue, EPICS test automation package was developed. Its purpose is to automate and facilitate testing of EPICS software on a wide range of hardware configurations and operating systems, to hide the differences of these hardware/software combinations and relive humans from complicated and error-prone manual testing. The system developed uses simple configuration and text-format files to describe the “test-scenarios”. It provides a flexible software framework to develop new test. Using this package greatly simplifies the testing process, reduces time and provides versatile environment for batch testing on a set of machines.

These studies and developments can be applied to any existent and future accelerator control systems to enhance the availability. Furthermore, the approaches described in the second chapter of this work are general and can be used in any environment to improve availability. Practical applications were developed for the EPICS system; the ideas and methods of expressed in this work can be adapted for systems other than EPICS.

Acknowledgments

I would like to express sincere gratitude to Prof. Shin-ichi Kurokawa at KEK, for being my advisor and giving me an opportunity of this work.

I would like to express my great gratitude to Dr. Kazuro Furukawa at KEK for his kind and invaluable advices and continuous support. This work would not be possible without helpful discussions and strong encouragement through the whole period of my research. Besides that, I would like to thank Dr. Furukawa, for his kind help and support of my life in Japan.

I am very thankful to Dr. Masanori Sato at KEK for helpful discussions, help on testing the EPICS redundant IOC and his kind support on numerous occasions.

I would like to express my profound thanks to Dr. Noboru Yamomoto at KEK for his useful advices and suggestions.

I wish to thank Dr. Matthias Clausen at DESY for kindly inviting me to DESY to conduct my research regarding the EPICS redundant IOC; Also I would like to thank Dr. Bernd Schoeneburg, Dr. Liu Gongfa at DESY for helpful discussions and explanations of the redundant IOC.

I am grateful to Dr. Shin Michizono for providing the ATCA hardware for my research.

I would like to thank Drs. Ned Arnold and Andrew Johnson at Argonne National Laboratory, for their kind help and invaluable advices with development of the EPICS test automation software.

Reference List

[1]: , INTERNATIONAL LINEAR COLLIDER REFERENCE DESIGN REPORT ILC, Global Design Effort and World Wide Study,
[2]: T. Himel , AVAILABILITY AND RELIABILITY ISSUES FOR ILC WEYAB02 Proceedings of PAC07, 2007 http://accelconf.web.cern.ch/AccelConf/p07/PAPERS/WEYAB02.PDF
[3]: Park Eung-soo, Faults and Reliability in the Operation of PLS , 2009 http://www.triumf.info/hosted/ARW/index.htm
[4]: J.F. Lamarre, Performances of the Soleil Synchrotron Light Source Operation , 2009 http://www.triumf.info/hosted/ARW/index.htm
[5]: KEK internal Documentation, Linac availability statistics , 2008
[6]: Furukawa, CONTROL SYSTEM OF THE KEKB ACCELERATOR COMPLEX ICALEPS2007, 2007 http://www-linac.kek.jp/linac/linac-paper/2007/ical2007-furukawa-kekb.pdf
[7]: J. KISHIRO, Control systems of the KEK accelerator complex PCAPAC2000, 2002 http://adweb.desy.de/mcs/pcapac2000/Proceedings/ID/144/ID144.pdf
[8]: Service Availability Forum, http://www.saforum.org/
[9]: Matthias Clausen, Gongfa Liu, Bernd Schoeneburg, REDUNDANCY FOR EPICS IOCS MOPA03 Proceedings of ICALEPCS07, 2007 http://accelconf.web.cern.ch/AccelConf/ica07/PAPERS/MOPA03.PDF
[10]: A.Kazakov, USING EPICS REDUNDANT IOC IN UNIX ENVIRONMENT TPPA31 Proceedings of ICALEPCS07, http://accelconf.web.cern.ch/AccelConf/ica07/PAPERS/TPPA31.PDF
[11]: EPICS 3.14.10 Release Notes, http://aps.anl.gov/epics/base/R3-14/10-docs/RELEASE_NOTES.html
[12]: Matthias Clausen et.al. , EPICS – FUTURE PLANS FOPA02 Proceedings of ICALEPCS07, http://accelconf.web.cern.ch/AccelConf/ica07/PAPERS/FOPA02.PDF
[13]: Ralph Lange, CA Gateway Update , http://www-csr.bessy.de/control/SoftDist/Gateway/download/GatewayUpdate.pdf
[14]: The OpenHPI Project: Sysfs Plugin, http://openhpi.sourceforge.net/manual/x415.html
[15]: EPICS home page, http://aps.anl.gov/epics
[16]: Paul Gibbons, Automated Testing Presentation to EPICS Collaboration Meeting 2007, http://www.aps.anl.gov/epics/meetings/2007-04/e138/TDI-CTRL-PRS-050%20EPICS%202007%20Automated%20Testing.pdf
[17]: J.F. Skelly and J.T. Morris, Design, Evolution and Impact of the AGS/RHIC Control System , http://accelconf.web.cern.ch/AccelConf/ica99/papers/wc1p05.pdf
[18]: , Reliability Theory Accelerator Reliability Workshop 2009, 2009 http://www.triumf.info/hosted/ARW/index.htm
[19]: Doug Preddy, Reliability Definition , 2009 http://www.triumf.info/hosted/ARW/index.htm
[20]: Jeffrey O. Hill, EPICS R3.14 CA Reference Manual,
[21]: S. Lewis, Overview of the Experimental Physics and Industrial Control System , EPICS http://csg.lbl.gov/EPICS/OverView.html
[22]: John L. Dalesio, Leo R. Dalesio, IOC Redundancy Design Doc Internal report, 2005
[23]: Bernd Schoeneburg, “API for the Redundancy Monitor Task Internal report, 2006

Reference List

Illustration Index

A. Appendix

A.I. Files

- RIOC source, available from DESY or from http://burdock.linac.kek.jp/kek/files/projects_02_Sep_2009.tar.bz
- rmt.apps (examples, redundant gateway etc.) are available here http://burdock.linac.kek.jp/kek/files/rmt_apps_02_09_2009.tar.bz2
- EPICS TEST automation package is available from github git://github.com/akazakov/epicstest.git or from (only rubyscripts) http://burdock/kek/files/rubyStuff2_02_09_2009.tar.bz2
- ruby scripts and mrksoft are here: http://burdock.linac.kek.jp/kek/files/soft-test_02_09_2009.tar.bz2

A.II. Building RIOC

- download and build EPICS BASE
  - Download EPICS base distribution from http://aps.anl.gov/epics/download/base/baseR3.14.11-pre1.tar.gz
  - and unpack it to ~/epics/base-3.14.11-pre1 directory
  - setenv EPICS_HOST_ARCH linux-x86
  - make
- get the source of redundancy software from desy
  - put it into ~/projects/redundancy/11august2009/iocRedundancy
  - edit configure/RELEASE and set EPICS_BASE to ~/epics/base-3.14.11-pre1
  - make

- create an application needed to be made redundant
  - Edit configure/RELEASE and set REDUNDANCY to the directory where you installed the iocRedundancy:
    - REDUNDANCY = /path/to/iocRedundancy/version
  - Edit your application Makefile:
    - Add 'rmt.dbd', 'cce.dbd' and 'snlexec.dbd' to your appname.dbd:
    - appname_DBD += rmt.dbd
    - appname_DBD += cce.dbd
    - appname_DBD += snlexec.dbd
  - Add 'rmt', 'cce' and 'snlexec' to the list of libraries which will be linked with your application:
    - appname_LIBS += rmt
    - appname_LIBS += cce
    - appname_LIBS += snlexec

Now your IOC is redundant. Consult the example configuration files from RIOC distribution. Enjoy.

A.III. Building load-balancing redundant CA gateway

- build the gateway
  - download and unpack epics extensions directory from http://aps.anl.gov/epics/extensions/configure/index.php
  - download and unpack gateway distribution into extentions/src
  - apply patch gateway2_0_3_0_load_balancing.patch (in rmt.apps: http://burdock.linac.kek.jp/kek/files/rmt_apps_02_09_2009.tar.bz2)
  - make

- go to *rtm.apps/run.bananas* to consult the *master.conf* and *slave.conf*
  - change the IP addresses to reflect you configuration
  - modify *start_master_lbgw.sh* and *start_slave_lbgw.sh* files to reflect your configuration
- run those scripts on master and slave.

Load balancing gateway should be running now.

Enjoy.

A.IV. Ca gateway implementation notes

According to gateway's nature it feels to have both of them up and running all the time. This will allow to load balance them seamlessly to clients. Assume that both gateways has the same configuration and in terms of network availability and access rights. So it means that both gateways can access the same IOC and PVs on these IOCs. In that case the working scheme might be very simple and the amount of synchronization data is minimal. Since version CA_V411 of channel access protocol, response to CA_PROTO_SEARCH request can have a server IP address. Let one of the Gateways be Master and the other one is Slave. First after initial startup gateways "handshake" and "decide" who is the Master. And from that moment gateways do not need to exchange any data except "heart-beat" and "health-status". And all search requests are answered by Master gateway. And master decides which IP address to put into the reply. It may be very simple as round-robin, or change the IP every next time. And obviously RMT can carry the function to make Master/Slave decisions, monitor connection status and probably some other parameters.

Step 1: Redundancy without Load-Balancing

rmtScriptDriver

The function of the driver is to run predefined external command when RMT changes state. It is very simple driver, and only start, stop, startUpdate, getStatus functions are implemented. But it is enough. It is implemented as a library (as RMT & CCE are). So the procedure of linking the driver to your source code is similar. Please refer to RMT & CCE Porting to *nix for more details. In short, you need these lines in Makefile:

- rmtStarter_DBD += rmt.dbd
- rmtStarter_DBD += rmtScriptDriver.dbd
- rmtStarter_LIBS += rmt
- rmtStarter_LIBS += rmtScriptDriver

To start configure and register driver use the following iocsh command: *rmtScriptDriver start_script stop_script*

Where start_script and stop_script are the executables you wish to run Master/Slave state change. Inside the driver they are executed via system(const char *string), which invokes the system shell with the parameter string. And this function waits until shell finishes. If it is not the desired behavior, replace it with something like execve().

start_script and stop_script

At first it was thought to block incoming CA_SEARCH request. But it appeared, that caGateway does create its internal PV variables (and does connection to the actual IOC) upon receiving search request. And when we block all the searches, caGateway then will deny all our direct connections to TCP:5064. Thus it was decided to block caGateway replies. To do this we invoke simple shell scripts when state changes. But there is one thing we have to remember. When SLAVE rmt is exited, it leaves DROP firewall rule, it's nothing bad, until you remember about it. First of all we create separate "chain" and redirect all desired traffic into it.

```
#!/bin/sh /script/nc_iptables -N RMT /script/nc_iptables -A OUTPUT -p udp --sport 5064
-j RMT
```

Now we can add/delete rules to that chain, and even flush it without affecting other firewall rules which might be present. And here is the start script:

```
#!/bin/sh /script/nc_iptables -F RMT
```

And the stop script:

```
#!/bin/sh /script/nc_iptables -A RMT -p udp --sport 5064 -j DROP
```

Master/Slave versions of the scripts are the same. But to run iptables on Linux you have to be root. So overcome this problem, we'll use a simple wrapper (which I found on the internet):

```
#include <errno.h> #include <stdio.h> #include <unistd.h> #include <sys/types.h>
#define ALLOWEDUID      15284 static char cmd[] = "/sbin/iptables"; static char
*smallenv[] = { "PATH=/sbin:/bin:/usr/bin", NULL }; main(argc,argv) int argc; char
*argv[]; { uid_t realuid = getuid(); if(realuid==ALLOWEDUID) { char *oldargv0 =
argv[0]; int e_errno; char buf[1024]; extern int errno; argv[0] = cmd; setuid(0);
execve(argv[0],&argv[0],smallenv); e_errno = errno; setuid(realuid); sprintf(buf,
"%.255s: %.255s", oldargv0, argv[0]); errno = e_errno; perror(buf); exit(255); }
setuid(realuid); fprintf(stderr, "%.255s: Permission denied\n", argv[0]); exit(254); }

And then: gcc nc_iptables.c -o nc_iptables chown root:kryo nc_iptables chmod 4110
nc_iptables
```

So now if the user belongs to group kryo, and UID == ALLOWEDUID he'll be able to execute iptables with root privileges. As you could notice this wrapper functionality is quite similar to sudo. Which also can be used.

Step 2: Load-balancing

This step required some modifications to the source of caGateway. And some kind of interface between caGateway and was introduced. RMT Changes: RMT's funtionality was extended to support sending signals to external process when the partner becomes alive/not alive. This functionality works only on Master side. To turn this functionality on, add the following string to startup script: rmtSignalPIDSet PID OK_SIG FAIL_SIG Where PID is pid of external process (so obviously it means you have to start caGateway first). OK_SIG is sent when the partner is up and healty, and FAIL_SIG is sent when the partner is healthy. We assume the partner being healthy if:

- RMT has connection with it
- All Interfaces are up
- No failed drivers are present

Also second “GLOBAL ETHERNET” check was added. It is configured with these options. System2IP=”1.2.3.4” System2Port=”1234” If they are not present, second check is not used. And logic of RMT was changed accrdigly. If one of the global checks fails on Master it switches to Slave ONLY if BOTH global checks are OK on Slave. caGateway changes: -partner_ip -partner_alive_sig -partner_dead_sig command line options have been added. First is partner IP address. Other two are signals to catch when the partner is alive/dead. At startup it is assumed that the partner is dead. Only after receiving correct signal load-balancing starts to work. And when it works every other search reply contains the partner’s IP address.

A.V. RIOC on ATCA