2005

Design and Implementation of a Non-Proprietary Campus Energy Management and Control System (EMCS)

Stefan Newbold
University of Nebraska-Lincoln, snewbold@unl.edu

Lalit Agarwal
University of Nebraska-Lincoln, lagarwal2@unl.edu

Published in IEEE International Conference on Electro Information Technology, 2005; doi: 10.1109/EIT.2005.1627032

Design and Implementation of a Non-Proprietary Campus Energy Management and Control System (EMCS)

Stefan Newbold, BSME, MSCS, P.E.
Software Engineer
942 North 22nd Street
Lincoln, NE 68588-0605
University of Nebraska – Lincoln
(402) 472-4852
snewbold@unl.edu

Lalit Agarwal, BSME
Software Engineer
University of Nebraska – Lincoln
lagarwal2@unl.edu

Abstract

The Energy Management and Control System (EMCS) used at the University of Nebraska - Lincoln (UNL) is unique in that system hardware and software has been developed primarily in-house. UNL has a successful track record with this approach stretching back more than twenty years.

This paper presents an industry experience report describing the high-level design and development of the latest version of this EMCS. This system is now being deployed on campus.

We discuss aspects of our EMCS that enhance usability, fault tolerance, and security. Our system is unique in that it was primarily developed using non-proprietary, open-source software building blocks and software construction tools. This approach provides a framework for potential collaboration with others who are interested in expanding this system beyond UNL.

1. Introduction

For the most part, University of Nebraska – Lincoln (UNL) students, staff and faculty are unaware of campus Heating Ventilating and Air Conditioning (HVAC) Systems unless of course they are too hot or cold. The HVAC systems within all major buildings at UNL are monitored and controlled by a networked, digital control system, which is commonly referred to as an Energy Management and Control System (EMCS).

UNL's EMCS is operated and maintained by the Department of Facilities Management. Our system is unique in that the majority of the field hardware and system software has been designed in-house since the early 1980s. Computer technology has changed a great deal since then. However, the reasons for pursuing an in-house EMCS solution remain unchanged. Key among these is the proprietary nature of commercial EMCS products. It is still the norm that once facility owners have made the decision to use a particular vendor, they are then very much locked into that relationship if they want to maintain a high degree of consistency and interoperability within their system. As a result, owners are forced into an environment where they are never entirely sure that they are seeing fair market prices for future equipment purchases and service contracts from the vendor. Other reasons for pursuing an in-house approach include flexibility, standardization of equipment, extensibility and most importantly, lower installation costs.

An EMCS has two main purposes: to control building systems and to conserve energy. We accomplish this through the manipulation of four kinds of real world, input/output interface objects, commonly referred to as "points."

- Digital Inputs (DI): These represent discrete (0 or 1) inputs to the system and correspond to wall switches, limit devices, etc.
- Digital Outputs (DO): These are the counterpart to DI points and represent such devices as equipment starters, two-position dampers, etc.
- Analog Inputs (AI): These represent non-discrete or continuum type inputs such as temperature sensors, pressure sensors, etc.
- Analog Outputs (AO): The counter part to AI points, used for modulating dampers, equipment speed control, etc.

System "intelligence" is provided by two software abstractions called loops and control blocks. Loops encapsulate a simple Proportional / Integral / Derivative (PID) control algorithm [7] whereas control blocks are small computer programs written in a domain specific language [20]. Control blocks are responsible for logic decisions and the overall manipulation of system input / outputs, loop objects as well as other control blocks.

The underlying mechanisms and software processes that comprise the EMCS are distributed between operator workstations, servers, building field computers, and thousands of networked embedded systems located in individual campus buildings.

This paper presents an industry experience report, which outlines the high-level architecture, design, and development of the software processes on each of the major hardware platforms that comprise the EMCS. We discuss some of the design imperatives as well as the trade-offs we faced. We discuss aspects of our system, which enhance usability, reliability and security. Finally, we illustrate how the system was designed and constructed using predominantly open-source [14] software building blocks and development tools.

The remainder of this paper is organized as follows. First, we present a high-level architectural view of the EMCS. We focus on the different hardware platforms that comprise our system and the manner in which these platforms are networked together. Next, we discuss the overall software architecture. We conclude with a summary of what we have accomplished and identify areas for future development and improvement.

2. Architectural View

This section presents a high-level architectural view of the EMCS, which focuses on the different hardware platforms that comprise the system and how these hardware platforms are networked together.

Figure 1 shows the overall architecture of UNL's EMCS. User interaction with the system occurs primarily via the EMCS user interface implemented on PC workstations. Workstation software is presented in Section 4.

PC workstations communicate with the EMCS servers via TCP/IP [4], allowing the system to be accessed from any location that has internet connectivity. We encrypt all packets between the user workstations and the server using a 64-bit SSL protocol [15]. This provides added security and prevents system tampering. Security is also enhanced via a dedicated firewall running Astaro Security Linux [1].

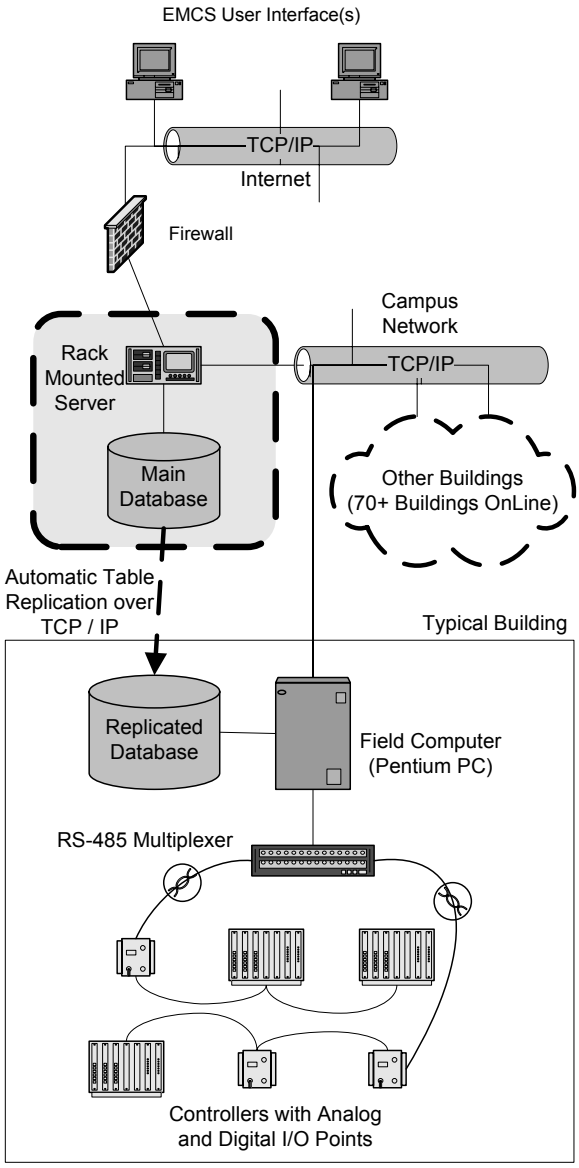

Figure 1: UNL EMCS Application Architecture

The EMCS servers are all rack mounted, Pentium Xeon machines running Red Hat Linux "Fedora" [22]. In order to provide some workload sharing, separate servers are dedicated to the following tasks:

- **Application Server:** Runs various EMCS processes that will be discussed in Section 5.
- **Data Base Server:** Is equipped with a RAID Level 5 [16] array and hosts the EMCS database. We use mySQL [19], a popular open-source database.
- **Web Server:** Provides graphic files to EMCS workstations via http requests. Our graphics package is discussed in Section 4.6.

EMCS servers communicate with building field computers over the campus network backbone using TCP/IP.

Each campus building is equipped with a single Field Computer (FC). The FC is a single-board, Pentium 3 computer plugged into a passive backplane, which along with a power supply and hard disk, are installed inside a wall-mounted cabinet.

The operating system for all FC's is Red Hat Linux, version 9.0. A mySQL database is also installed in order to be able to store a replicated portion of the EMCS database. This allows us to keep track of all directly connected controllers. EMCS software processes for the FC's are discussed in Section 6.

EMCS Controllers are constructed using 8 or 16 bit microcontrollers manufactured by Atmel [2] or Rabbit Semiconductor [21]. They are capable of supporting AI/AO, DI/DO, Loop and CB objects (See Section 1). Controller software is presented in Section 7.

Communication between the FC and its associated controllers occurs via RS-485 [9] communication links running at 19,200 or 57,600 baud. These links are connected to an in-house designed multiplexer board, which is installed in the same backplane as the FC. Our multiplexer subsystem can accommodate up to 2048 controllers spread over 32 links per FC.

Referring back to Figure 1, there are three communication "hops" in our system.

1) User workstation to EMCS server (TCP/IP and SSL)
2) EMCS server to Field Computer (TCP/IP)
3) Field Computer to Controller (RS-485)

Even though different protocol stacks are used at each hop, the application layer protocol at each hop is the same. This protocol was also developed at UNL and is designed around an attribute / data-tagging scheme thereby making it very extensible.

3. Software Technologies and Design Approaches

This section discusses some of our high level software design technologies, approaches and trade-offs as well as some issues that are orthogonal to all of our software processes.

3.1 Technologies

Design of the new system software began during the spring of 2001. One of the most basic issues we had to face involved the choice of software technologies to be incorporated into the design.

Being an academic institution with limited monetary resources for software purchases, one technology paradigm we embraced very early was the use of open-source building blocks and software construction tools. The Linux Operating System [18] and mySQL Data Base [19] were two components we gravitated towards immediately. Since these components have matured and achieved more market penetration over time, we feel that these were wise choices.

At the beginning of our design process, we investigated the trade-offs between a *lightweight* and *heavyweight* User Interface (UI). A *lightweight* user interface is typically developed to run within a web browser. In contrast, a *heavyweight* UI is developed using one of several UI frameworks.

We eventually abandoned the *lightweight* client approach for several reasons.

- Since our user base is not large, the ease of installation and maintenance benefits associated with a *lightweight* user interface were minimal.
- It is more difficult to construct a feature rich UI with a *lightweight* approach.
- We were concerned about the latencies associated with continuously pulling all information and components to populate the UI from the server.
- The same cross-platform capability of *light-weight* approach can be achieved if one uses a Java native UI framework

Our UI was ultimately designed using Java AWT / Swing classes as our framework. However, if we were to make the same choice today, we would strongly consider using Eclipse Standard Widget Toolkit (SWT) [10] classes since this alternative GUI framework has built a lot of momentum in the last few years.

Since we were familiar with C/C++ and the fact that our processes need access to low-level operating system calls, we wrote all server and field computer processes using these two "sister" languages. For code development, we used the GNU [5] compiler and KDevelop [12] Integrated Development Environment (IDE).

3.2 Design Approaches

Once we decided on the technologies we would use for our system, we had to consider the trade-offs of various design approaches. One of these dealt with whether or not we should partition our server and field computer applications into multiple processes. For example, one approach would be to design a monolithic application for the server and then dynamically create a new instance of this application (i.e., "fork" system call) for every user login. The main advantage of this scenario is that no inter-process communication would be required.

Ultimately, we decided to use a modular approach by splitting our server and field computer applications into several processes. These are discussed in Sections 5 and 6. Partitioning the application made it easier to debug problems, add new features as well as allowing team members to concurrently write and test different process modules.

The concept of fault tolerance is of particular importance for this application domain. We therefore designed the server and field computer processes so that they will continue to run even if other processes are shut down, suffer a segmentation fault, etc. In addition, all inter-process communications are designed so that these connections are automatically re-established after any process fault events.

One very "Unix-like" concept we incorporated into our system was the use of a plain text configuration file to initialize process' operating parameters on start-up. This makes system tuning and troubleshooting much easier than if we had "hard-coded" such parameters into our source code.

4. User Interface

This section discusses the design approach as well as some of the major features of our EMCS User Interface (UI). As previously discussed in Section 3.1, the UI was built with the Java AWT/Swing Framework using the Netbeans Integrated Development Environment (IDE) [13].

4.1 User Interface Design Approach

We chose a multiple document interface strategy for our UI. This approach presents the user with one main parent frame as a "container" for any number of child frames. Our child frames provide various views of the system and are discussed in subsequent sub-sections.

Our design includes several reusable components. Such reuse was enhanced by using the Model View Controller Design Pattern [6] for all major UI frames.

In order to provide true end-to-end EMCS packet routing, all packets sent by the UI into the system are addressed to a particular EMCS object or process using a hierarchical, numeric object ID. We maintain a series of tree-like data structures to map object acronyms (i.e., text) to these object IDs. Acronyms and object IDs are retrieved from the server and then stored locally at the user workstation to provide quick lookups.

The UI executable is compiled into a single Java archive (jar) file and is placed on our web server where users can automatically retrieve the latest version using Java "Web Start" [8] technology.

The following sections discuss some of the individual child frames of our UI and the respective system views that they provide.

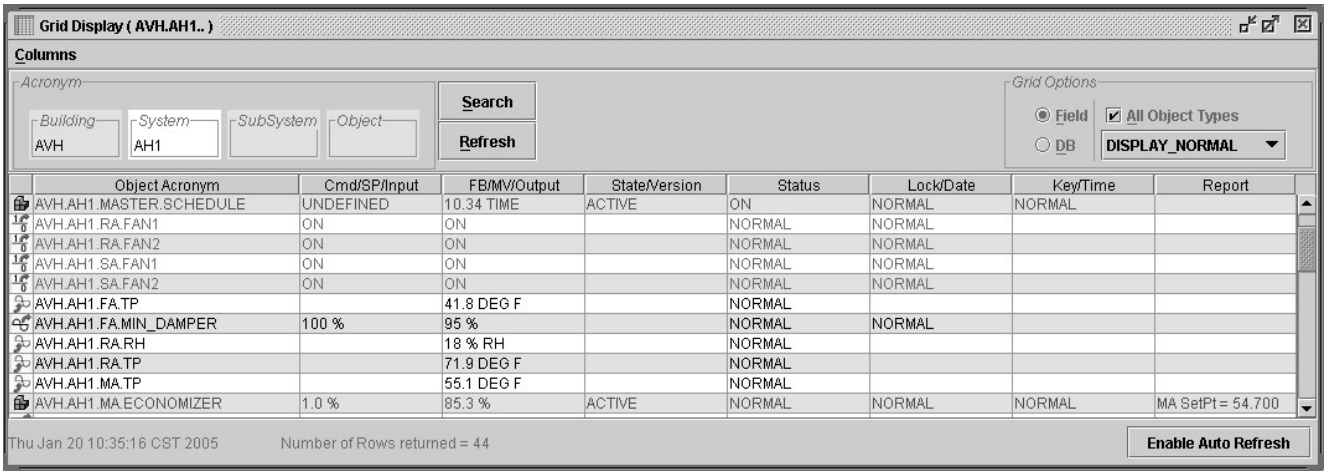

Figure 2: User Interface Grid Display Frame

4.2 Grid Display Frame

The Grid Display (GD) presents both real time and static object data in a table format. Our GD is unique in that we have designed it so that a user can quickly drill down to any subsystem or individual object without having to click on a number of intermediate tables or views. For example, the GD shown in Figure 2 shows a portion of the air handling system for a particular building. Alternatively, users can simultaneously view objects that span multiple systems and buildings on one single table.

This customization and flexibility is provided dynamically without forcing the user to predefine such table views.

4.3 Tree Display Frame

The tree display frame provides a hierarchical, hardware platform view of the system with the EMCS server and field computers at the root level. Expanding these nodes shows all resident objects or other connected controller hardware objects. EMCS objects resident on a particular hardware platform comprise the leaf nodes of the tree display.

4.4 Chart Display Frame

The chart display provides the capability to plot live data from a number of EMCS objects. We constructed this display frame with the aide of JFreeChart [11], an open-source plotting framework.

4.5 Distributed Control Language (DCL) Control Block Editor Frame

Control blocks encapsulate the operational logic for our system and were previously discussed in Section 1. They are written in a domain specific language called Distributed Control Language (DCL) [20]. Our DCL editor provides a utility for writing, compiling and debugging DCL code. This frame includes all the standard features one would find in any recent code editor including line numbering, syntax highlighting, etc.

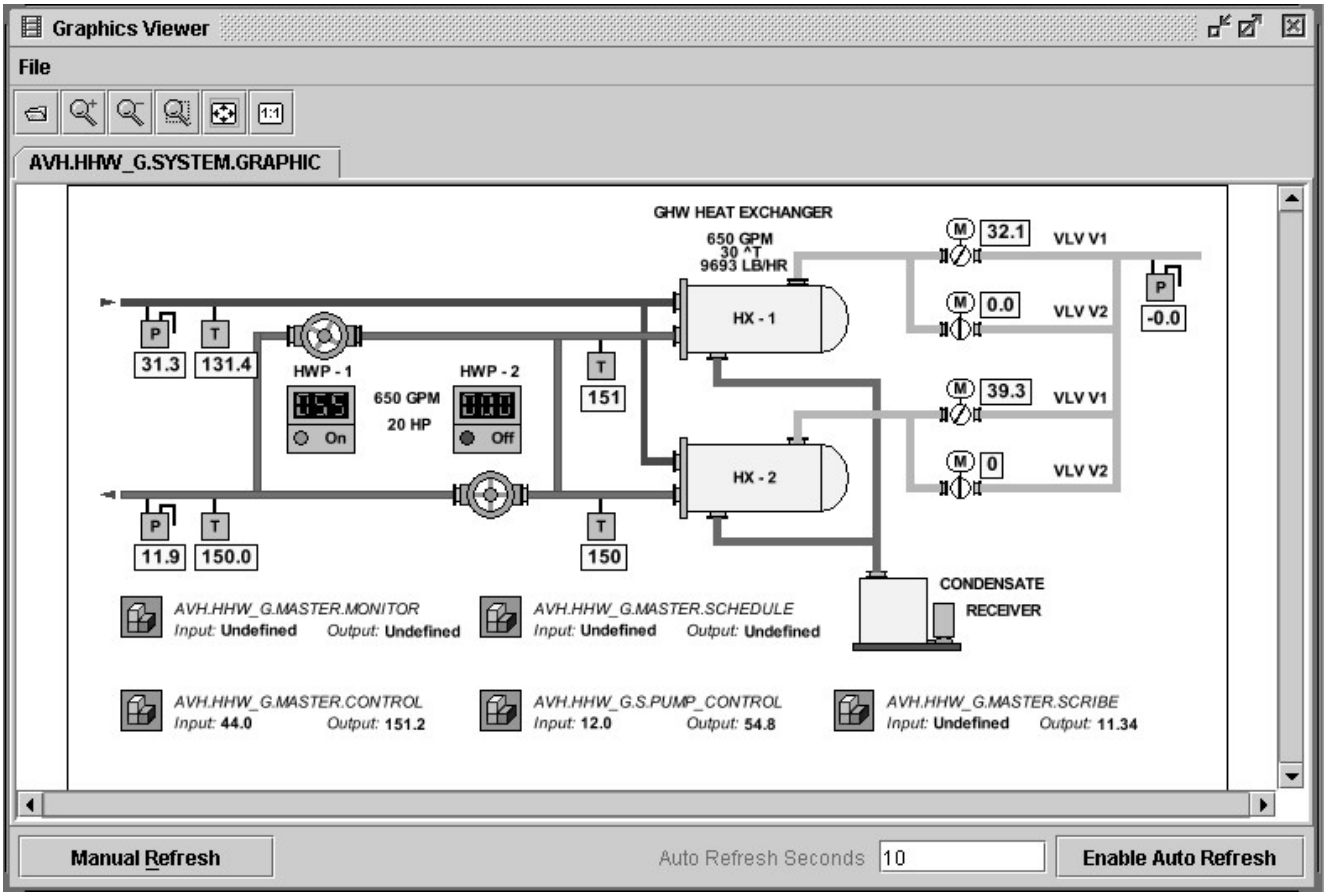

Figure 3: User Interface Graphics Viewer Frame

4.6 Graphics Composer and Graphics Viewer Frames

The graphics composer and graphics viewer frames are used to create and view real-time EMCS system values in a schematic format. To speed development of this aspect of our UI, we reluctantly broke with our "open-source" paradigm and purchased the JViews graphic framework by ILOG [17]. This framework was used for the development of these two child frames. However, it would not be difficult to remove the code for these frames from our application if we wished to make the entire UI "open-source."

Once a system graphic file is created using the graphics composer frame, the file is stored on our web server. When launching the graphics viewer frame, one or more graphic files are retrieved and then displayed. Figure 3 shows a typical graphic screen for one of our heating hot water systems. Users can control the system from these graphic screens via mouse clicks and other interactions.

4.7 Other Frames

Additional child frames that are available within our UI include:

- History, Trend, Report and Log Viewer Frames
- Chat Window Frame (allows logged-in users to chat)
- User Group Administration Frame
- Individual EMCS Object Definition and Real Time Data Frames

5. EMCS Application Server Software Components

This section presents an overview of the EMCS software processes resident on our application server. These processes are shown in Figure 4.

5.1 Field Computer Manager

The Field Computer Manager (FCM) maintains local sockets to each of the server resident processes as well as a TCP/IP socket to each of the system field computers. The primary responsibility of the FCM is to route EMCS packets between server resident processes and Field Computers.

When a local process or field computer connects and identifies itself, the FCM adds the client socket and its routing information to a hash map. The FCM checks all active sockets for packet activity in a round-robin manner. If an active socket has any packets, the FCM "drains" the socket for a pre-determined number of packets or until there are no more packets pending on that socket. This strategy is a compromise, which attempts to provide good throughput on continuous data streams, without starving other sockets.

If the hash map lookup for a client socket fails, the FCM takes the following actions.

- If the original packet was sent by another server process or user, an error is sent back to the original sender.
- Otherwise, the packet is handed to Garbage Manager (Section 5.7).

The FCM continuously checks all active sockets for communication error conditions or process disconnects and continuously updates its socket hash map accordingly.

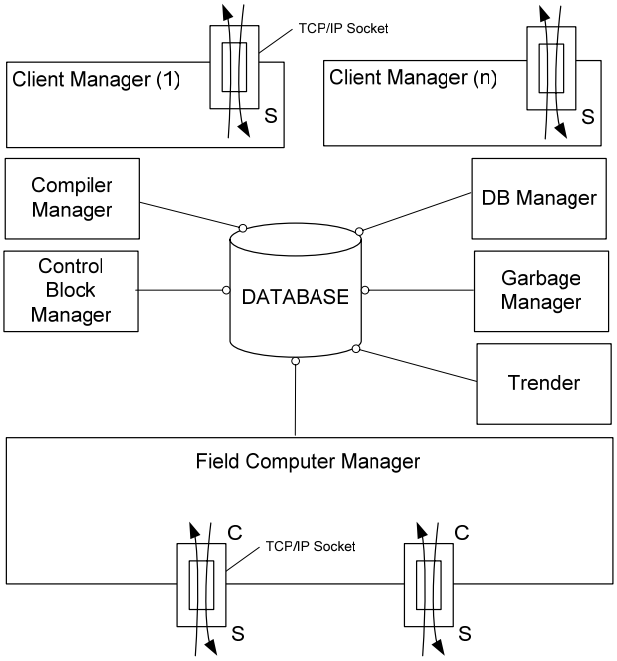

Figure 4: EMCS Application Server Software Components

5.2 Database Manager

The main task of the Database Manager (DM) is to provide a business logic layer between the user interface and the system database. The DM implements object creates and deletes as well as reads and writes of static definition data. The DM communicates with the system through a dedicated socket to the FCM.

All object create / read / write / delete requests are in the form of EMCS packets. The DM parses these packets, performs any necessary business logic, and then generates appropriate SQL statements. Any parse, logic or SQL problems will generate an error packet, which is sent back to the requestor.

Depending on the nature of the original request, the DM may send packets addressed to a particular controller, EMCS object or another process. Since communication latencies become significant during packet exchanges with controllers, the DM maintains a packet registry that keeps track of all requests and associated responses or timeouts. This allows the DM to asynchronously process additional requests while responses to previous requests remain outstanding.

5.3 Compiler Manager

The Compiler Manager (CM) processes all requests related to control block source code (See Section 1). The CM's main responsibility involves compilation of user generated control block source code into byte code and then downloading this byte code to EMCS controllers or field computers. Due to the dynamic data structures required by the compiler (e.g., parse tree, symbol table), we spawn a new CM instance with each new compiler request via a "fork" system call. Thus, memory for all dynamic data structures are returned to the heap after the request has been fulfilled.

The CM communicates with the rest of the EMCS with local sockets in a manner similar to the Database Manager (DM).

5.4 Client Manager

The Client Manager (CLM) acts as the "gatekeeper" between user workstations and the EMCS. This process maintains a TCP/IP socket, which listens for user login request. During such a login request, the CLM attempts to authenticate the user. If this authentication is successful, the CLM "forks" to create a new CLM instance solely dedicated to a particular user login. If the user is not authenticated, an error packet is sent back to the user workstation and the CLM terminates the session.

The "forked" CLM maintains a dedicated socket to the Field Computer Manager for the duration of the user login. The CLM will terminate a user session after a specified period without any packet activity. This period can be set when a user is defined in the system can be modified anytime by a user with EMCS administration rights.

5.5 Control Block Manager

Control Blocks (CB's) provide the overall operational logic for the system and were briefly discussed in Section 1. When a CB is created, the user must define the location where the CB will reside (i.e., execute its code) among three possibilities:

1) The EMCS Server
2) A particular Field Computer
3) A particular Controller

The Control Block Manager (CBM) is responsible for executing the byte code associated with each resident CB and to process any I/O generated by these CB's. At the server level, all CB I/O involves sending out and receiving EMCS packets. The CBM acts as a proxy for this I/O on behalf of the CB.

CB's are executed by the CBM in a round robin fashion by dispatching CB opcodes for a given number of instructions or until the CB requires I/O. The CB is then placed at the end of the CBM's run queue. If a CB has an outstanding I/O request, the CBM will continue to place the CB at the end of the run queue until a response is received or the request times out.

5.6 Trender

In this application domain, it is often desirable to temporarily cache real time object data for the purpose of troubleshooting problems or erratic behavior. Typically, only the most recent few days worth of data is maintained. The Trender process is responsible for gathering live data, entering it in the database, and purging old data. This data can then be viewed at the EMCS operator workstation.

The Trender process at the server only gathers data for server resident objects (i.e., CB's). Trender processes are also present on field computers (Section 6) and are responsible for gathering data for all field computer resident objects as well as objects that reside on any connected controllers.

5.7 Garbage Manager

The primary responsibility of the Garbage Manager (GM) is to handle packets that the Field Computer Manager (FCM) is unable to resolve. Packet information is logged into a file and then dropped.

The GM has an important additional role which makes its name somewhat misleading. Control Blocks (CB's) can be written so that they send up real time object data for long-term energy use or performance analysis. These

types of packets are parsed by the GM and inserted into the EMCS database.

6. Field Computer Software Components

This section presents an overview of the EMCS software processes resident on each Field Computer (FC). These are listed as follows:

- Controller Manager
- Control Block Manager
- Trender

6.1 Controller Manager

The Controller Manager (CM) has two tasks: maintain communication paths and route EMCS packets. Communication to the EMCS server takes place via a dedicated TCP/IP socket. The FC communicates with connected controllers via a RS-485 [9] multiplexer board, which was designed in-house.

The CM routes packets between controller links, the EMCS server and other FC processes. Managing packet routing on controller links was a particular challenge due to the baud rate limitations and the fact that RS-485 can only provide a "half-duplex" communication channel. This means that only one device can be transmitting at any given time.

CM-to-controller communication uses the *master-slave* paradigm. The *master* (CM) always initiates communication, whereas the *slaves* (i.e., controllers) only transmit in response to a request from the *master*.

The CM maintains four EMCS packet priority queues for each RS-485 link. This allows us to prioritize packets that contain more important or urgent communication requests.

We also maintain a link map data structure within the CM that contains an entry for each connected controller. Each entry provides information regarding the controller's link number and baud rate.

The CM process was designed to be "self-healing." What we mean by this is that CM can re-establish the connection to the EMCS server after a network failure or server re-boot. In addition, it will adjust the link map as controllers are added or removed.

The CM contains four modules named:

- "Q"
- Finder
- PushPuller
- Poller

"Q's" role is to listen for EMCS packets on all local sockets (to other FC processes) and the single TCP/IP socket back to the EMCS server. "Q" routes packets to the correct socket or, if the packet is addressed to controller, "Q" will add the packet to the appropriate link priority queue. If a packet is addressed to a controller that does not have an entry in the link map, the packet is handed over to Finder.

Finder is responsible for discovering the baud rate of a particular controller and on which RS-485 link it resides. This discovery process is started by sending a packet addressed the controller we wish to find, down all RS-485 links at a particular baud rate. If the controller exists on one of the links and is communicating correctly, it will acknowledge our find request. Once the controller is found in this manner, the link map data structure is updated with the appropriated link number and baud rate.

If all the packets sent by Finder time out, we perform another iteration at the next higher baud rate. This process repeats until the controller is found, or all Finder packets have timed out at all baud rates. In the latter instance (i.e., "controller not found"), Finder sends an error packet back to the original requestor.

Pushpuller's task involves packet transmission and reception to controllers on all connected RS-485 links. Sending EMCS packets to individual controllers involves PushPuller draining the various link priority queues discussed previously and then transmitting the packet on the RS-485 communication link. Figure 5 is a UML Activity Diagram [6] illustrating the algorithm that PushPuller uses for packet transmission and reception for one particular link.

Note that if a request packet has been sent on a RS-485 link, we "lock" this link from further requests until a response is received or the packet times out. We compute individual packet timeouts dynamically based on packet size and the particular baud rate at which they are transmitted.

Poller's single function is to "poll" the controllers by periodically sending very short packets. This serves three purposes:

1. It allows us to synchronize controller date and time values to the current system time.
2. It allows controllers to send any packets they may have queued up since the last poll.
3. It provides a continuous mechanism for updating our link-map data structure. In effect, we are asking each

Controller the question "Are you there?"

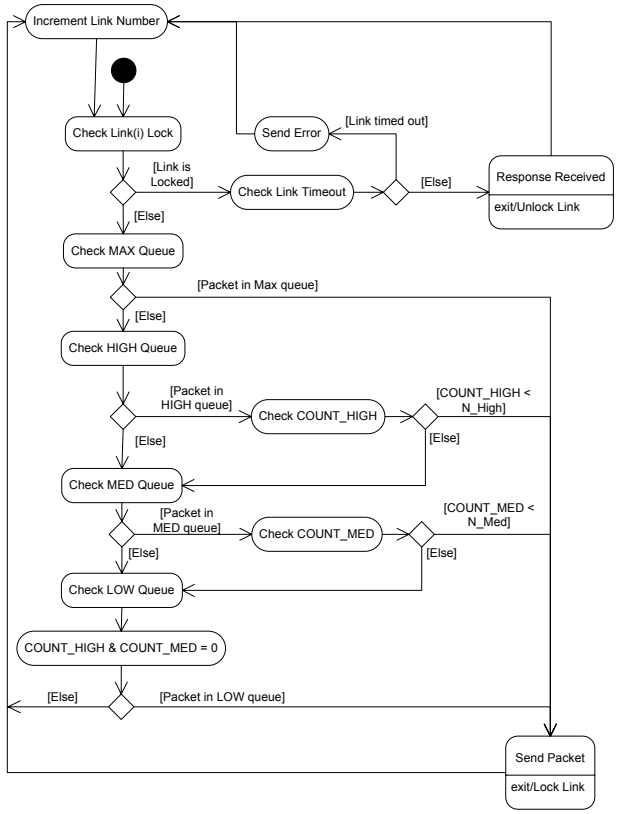

Figure 5: Activity Diagram for Controller Manager RX / TX on RS-485 Link

6.2 Control Block Manager and Trender

Other EMCS processes resident on a Field Computer includes the Control Block Manager and Trender. These are similar to the processes previously described for the EMCS application server in Section 5.

7. Controller Software Components

This section outlines the major software components that reside on the lowest rung of our system hierarchy. We do not go into as much detail as we have with the server or field computer processes since a great deal of the Controller software is tightly coupled to our embedded processors and its I/O peripherals.

Other than communication, the building controls domain does not really have any strict timing constraints as may be present in other embedded process control systems. We therefore elected not to incorporate any type of real-time operating system. An operating system would also have added some additional memory burdens on a platform, which is already resource, constrained. Our process scheduling strategy is very simple. We give equal priority to all controller tasks except communication, which is given higher priority and is entirely interrupt driven.

All controllers include modules for loop objects and interpreters for control block objects. This allows us to distribute our system intelligence down to the controller level, thereby making the overall system more robust and fault tolerant.

Figure 6 shows how our controller software modules were designed with a layered approach. All modules located in the middle layer are designed to be platform neutral so that they can be reused in future controller hardware platforms. In contrast, the lower software layer is tightly coupled to the actual hardware platform and is not portable.

8. Evaluation and Future Work

We have designed and developed a software solution for a fully functional campus Energy Management and Control System. Our system is designed so that it is fault tolerant, extensible and easy to maintain. Fault tolerance is further enhanced by the fact that all of our low-level controllers are able to support control blocks (Section 1). This means that communication outages will not affect basic system control functions.

Our Java based User Interface (UI) provides a feature rich environment and which is portable to other hardware platforms for which a Java Virtual Machine has been implemented. The overall system response time at our UI is equivalent to, or better than our first hand experience with other commercially available systems.

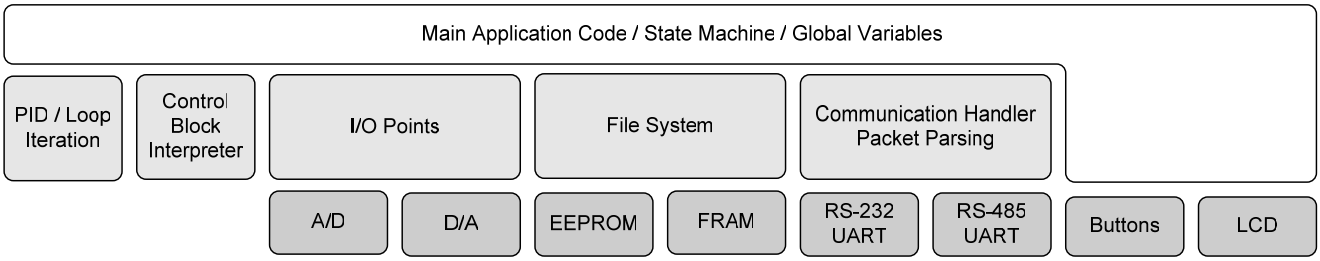

Figure 6: Controller Software Module Layering Scheme

Finally, the majority of our system was built using open-source building blocks and software construction tools. This makes it easier for others to adopt and expand upon our system.

We have identified a number of system enhancements and extensions we would like to implement.

- Card access capabilities. Our goal is to provide a campus wide, cost effective alternative to door keys. One innovation we are starting to work on is a combination room temperature / card access controller that would eliminate the need for two separate controllers for offices, classrooms and dormitory rooms.
- Better capabilities for system alarming. This will be of particular importance as we move into access control.
- UI code generation "wizards" that would generate control block code for simple tasks such as system scheduling.
- More UI customization capabilities, particularly with respect to the chart display frame (Section 4.4).

9. References

[1] Astaro Corporation, 3 New England Executive Park, Burlington, MA 0180. http://www.astaro.com.

[2] ATMEL Corporation, 2325 Orchard Parkway, San Jose, CA 9513, www.atmel.com.

[3] Booch, G., Rumbaugh, J., Jacobson, I. *The Unified Modeling Language User Guide*. Addison-Wesley, NJ, 1999.

[4] Comer, D. *Internetworking with TCP/IP Volume I: Principles, Protocols and Architecture*. Prentice Hall, NJ 07632, 1991.

[5] Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA 02111-1307, http://directory.fsf.org/devel/compilers/gcc.html

[6] Gamma, E., Helm, R., Johnson, Vlissides, J., *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, NJ, 1995.

[7] Honeywell Inc., *The Engineering Manual of Automatic Control*, Minneapolis, MN, 55408, 1988.

[8] http://java.sun.com/products/javawebstart

[9] http://www.bb-elec.com/tech_articles/rs485_basics.asp

[10] http://www.eclipse.org/swt

[11] http://www.jfree.org/jfreechart

[12] http://www.kdevelop.org

[13] http://www.netbeans.org/index.html

[14] http://www.opensource.org/docs/definition.php

[15] http://www.openssl.org

[16] http://www.storagereview.com/guide2000/ref/hdd/perf/raid/levels/singleLevel5

[17] ILOG, Inc., 1080 Linda Vista Ave., Mountain View, CA 94043. http://www.ilog.com/products/jviews

[18] Linux Online, Ogdensburg, New York, USA. www.linux.org

[19] MySQL Inc. 2510 Fairview Avenue East, Seattle, WA 98102, www.mysql.com

[20] Newbold, S., *Design and Implementation of a Distributed Control Language for a Campus Energy Management System*, Masters Thesis, University of Nebraska – Lincoln, Department of Computer Science and Engineering, 2003.

[21] Rabbit Semiconductor, 2932 Spafford Street, Davis, CA 95616-6800. http://www.rabbitsemiconductor.com

[22] Red Hat, Inc., 1801 Varsity Drive, Raleigh, NC 27606. http://www.redhat.com