SIMON: A DISTRIBUTED REAL-TIME SYSTEM FOR CRITICAL CARE PATIENT MONITORING AND EVENT DETECTION

By

Karlkim Suwanmongkol

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

August, 2001

Nashville, Tennessee

Approved:

Dr. Benoit Dawant

Dr. Gabor Karsai

ACKNOWLEDGEMENTS

I would like to express my gratitude to my academic and research advisor, Dr. Benoit Dawant for his determined support, encouragement, and guidance within and beyond the scope of this research project since I came to Vanderbilt University. Special thanks to my good friend and outstanding team member Patrick Norris for his tireless assistance to me and strong contributions to this project. Without their assistance and the regular discussions we had, the project would not have been at this stage. I appreciate Dr. Gabor Karsai for his excellent classes inspiring countless ideas to me, our fruitful discussions, and his precious time to be a second reader for this work. Further thanks go to Dr. Dan Lindstrom and the Neonatology Division of the Pediatrics Department for their financial support during the first two years. I thank Dr. Antoine Geissbuhler and Dr. John Morris for contributing their ideas and expertise to the project. Many thanks to Eric Manders for providing help with the legacy code and the equipment, Russ Waitman for his suggestions about the HPDA component, fellow Thai students, P'Pinij, P'Suchart, P'Siam, P'JJ, Arnon (fat C), and Putthipong (skinny C), who provided initial technical help and knowledge. I also thank all the other members of the Image Processing lab for providing a friendly atmosphere at work

My beloved mom, dad, and brother deserve tremendous credit, more than any words could express, for their love and care despite the geographic distance for many years. I sincerely appreciate my grandma, Gregoria P. Ignacio, my granduncle, Bernardo T. Pedere, and uncle Jim, James Reese, who always supported me since I came to the USA. Finally, I would like to thank all my friends and well-wishers who, knowingly or unknowingly, provided me with all the help I needed to complete this work.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

BACKGROUND AND INTRODUCTION

In the critical care unit, the real time monitoring of patients is an essential task. Patient Monitoring can be defined as: “Repeated or continuous observations or measurements of the patient, his or her physiological function, and the function of the life support equipment, for the purpose of guiding management decisions, including when to make therapeutic interventions, and assessment of those interventions” [1]. Thus, patient monitoring and management involve determining status of the patient, responding to events that may be life threatening, and taking actions to bring the patient to a desired state.

New bedside medical devices provide health care professionals with unsurpassed amounts of information for decision-making support. In addition to information provided by these bedside devices, health care professionals also rely on information such as laboratory data or demographic information coming from the hospital information system (HIS). Data from patient monitoring devices and sophisticated laboratory tests generates a flood of data that can be difficult for individuals to process and abstract in real time. This, in turn, can lead to sub-optimal decisions and therapeutic actions [2].

Interpreting this information requires the integration, correlation, and comparison of data acquired by a variety of sources. Patient monitoring system needs to capture data from all bedside devices automatically. Therefore, computer programs must be able to acquire data from all of these devices directly. Unfortunately, these bedside monitoring devices are often stand-alone and use proprietary communication protocols, making their interconnection difficult. Although, an IEEE standards committee, IEEE 1073 [3], has been working since 1984 on a standard for medical device data communication in critical environments called the Medical Information Bus (MIB), major medical device manufacturers still do not adhere to the standard. Even today, the lack of interconnection remains a major obstacle to the development and implementation of intelligent monitoring systems.

Thus, new solutions are needed to integrate the data from bedside devices and hospital information systems, manage and process the flow of information and provide efficient and reliable decision support tools. Ideally, a real-time patient monitoring system should be able to

acquire physiology data from all bedside monitoring devices as well as relevant information from the HIS and provide real-time and systematic integrated information for medical practitioners. This monitoring system should also provide ways to access the information remotely, and notify users when critical events are detected.

Hence, from the above discussion a patient monitoring system should be able to:

1. Acquire physiological data (from bedside devices)

1.2. Acquire information from the HIS such as demographics or lab data.

1.3. Store a large amount of data, organize it, and provide information to the medical practitioners that can be used for decision support.

1.4. Integrate and correlate data from multiple sources.

1.5. Automatically detect adverse events and notify care providers.

Over the years, a number of systems have been developed to address problems faced by clinicians in critical care environments. These include, among others (see [4] for a more complete review), model-based event detection [5], planning and critiquing [6], or closed-loop control of bedside medical devices, such as ventilators [7] and infusion pumps [8]. But these systems were developed with specific applications in mind and are difficult to scale up. They are difficult to use in a wide range of patient monitoring applications. The main objective of this work is to develop a flexible and scalable architecture for real-time patient monitoring in intensive care units.

SIMON

SIMON, an acronym for **S**ignal **I**nterpretation and **MON**itoring, is a system being developed at Vanderbilt University, in the School of Engineering in collaboration with the Vanderbilt University Medical Center. SIMON is designed to acquire, integrate and process information acquired from bedside-devices, medical staff, and the hospital information system. The main goal of this project is to design, develop, implement, and test software and hardware architecture for intelligent patient monitoring. A requirement is that this architecture be easily fielded in a variety of critical care units and configured for a wide range of monitoring applications.

Prior Architecture

An initial architecture of SIMON was designed and implemented in 1994 [9, 10, 11, 12]. The system had been running in the Coronary Care Unit and the data from the unit were published on the web near real-time [10,13] from February, 1996 to August, 1997. Development of the SIMON-ICU prototype comprises three parts: SIMON-RTS, SIMON-ART, and SIMON-WEB. Each part is discussed below.

SIMON-RTS

The main component of this version of SIMON is the real-time monitoring system (SIMON-RTS). The functionality of the monitoring system is separated into modules. Modules communicate with each other through Unix Inter Process Communication (IPC) [14]. A Client-Server architecture is used to organize the modules. The main modules are the Data Acquisition module [15], Feature Extraction module and the Monitoring Supervisor. The Data Acquisition module acquires the raw data from bedside devices through their serial port interface (RS-232). The second module is the Feature Extraction Module. The Feature Extraction Module implements all data validation, sensor fusion and feature extraction operations under the control of the Monitoring Supervisor. The Monitoring Supervisor controls the feature extraction module and is responsible for high level monitoring directives in a data abstraction strategy.

SIMON-ART

The functionality of the SIMON-ART system is to publish monitored data collected by SIMON-RTS on the web. The ART system acquires data from SIMON-RTS via FTP transfers from the bedside computer and then graphs the monitored data as GIF images at various time resolutions, and publishes them on the Web along with HTML pages containing laboratory data and physician orders.

SIMON-WEB

SIMON-WEB is a Java application that integrates data collected by SIMON –RTS and the hospital information system (CIS). SIMON-WEB is a first attempt at providing integrated information to the medical practitioners. SIMON-WEB displays graphical data transferred from

the SIMON-ART by way of FTP transfers as well as information from the clinical laboratory and physician order-entry systems. The system allows users to annotate the data using a point-and-click or free-text interface. The nurse's annotations help in the electronic capture of significant clinical events that cannot be sensed automatically; consequently, it can be used to assist in developing better event detection and artifact rejection algorithms.

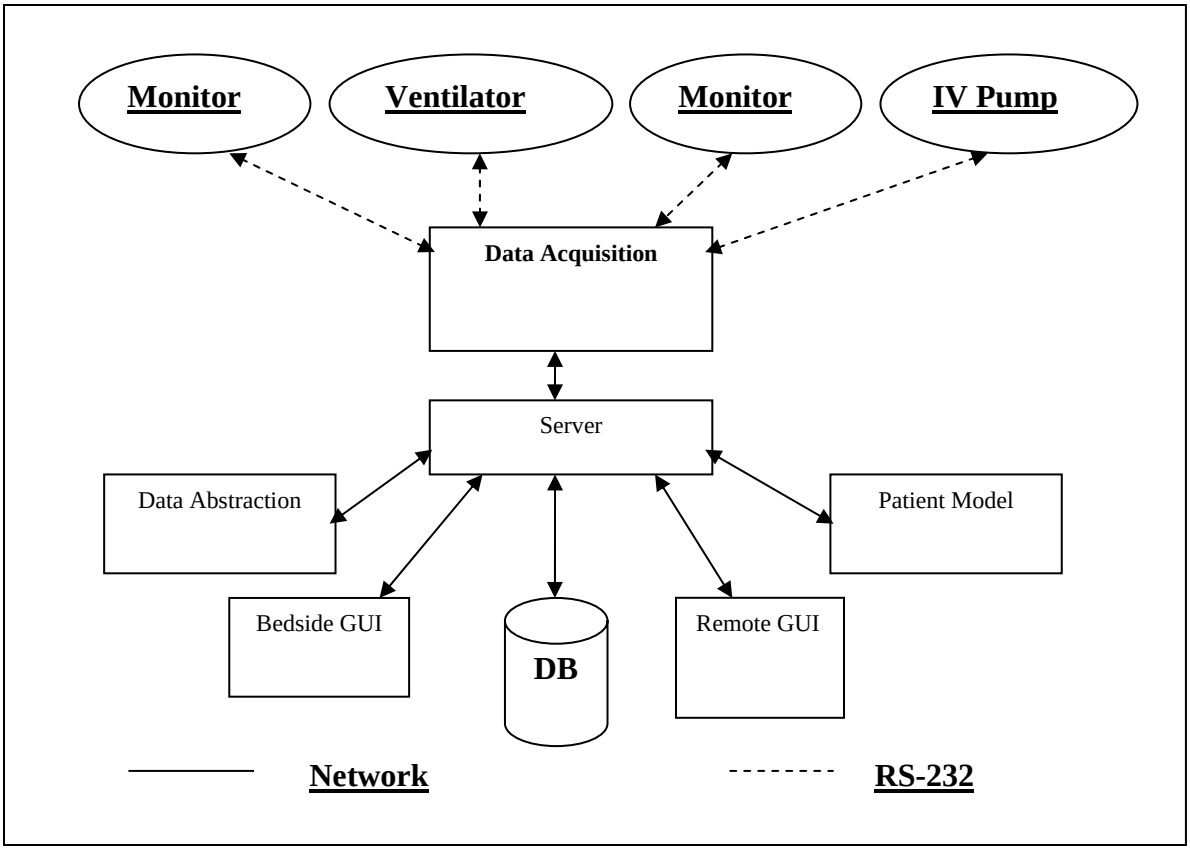

Figure 1: Architecture of SIMON-RTS [10]

Evaluation

This version of the SIMON architecture was fielded in the Coronary Care Unit (CCU) of Vanderbilt University Medical Center (VUMC). SIMON-RTS and SIMON-WEB ran on a bedside workstation that exported data to SIMON-ART, which, in turn, published the data securely on the VUMC intranet. Figure 3 shows the architecture of the SIMON system in the CCU.

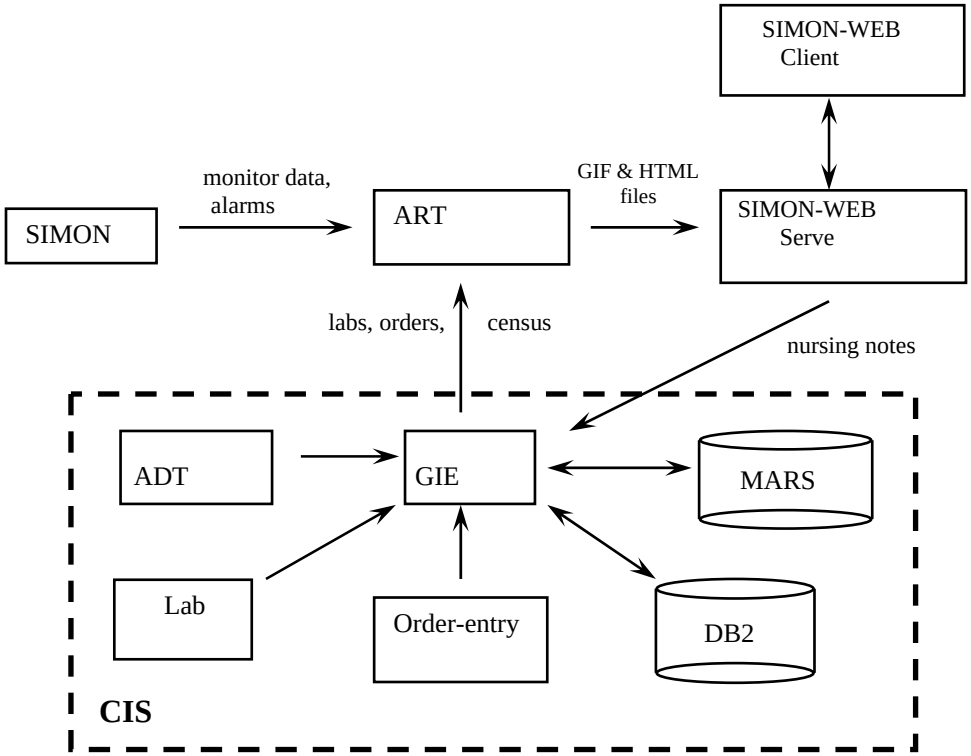

Figure 2: SIMON-WEB system architecture and relationship to the clinical information system (CIS) [10]

In this version of the system, the modules did provide rigorous abstraction of device communication and data collection functions. However, this abstraction adds considerable complexity to the system. Tight coupling between the Server module and the data collection/analysis modules makes the system difficult to expand for new devices and new beds. In addition, the skills and resources available to the project and the local computing infrastructure at the time were not well-suited to Solaris x86. Windows NT seemed a more appropriate choice for these and other reasons, and a large portion of the code needed to be rewritten in order to run the system on this platform. These considerations lead to a complete overhaul of the system [16].

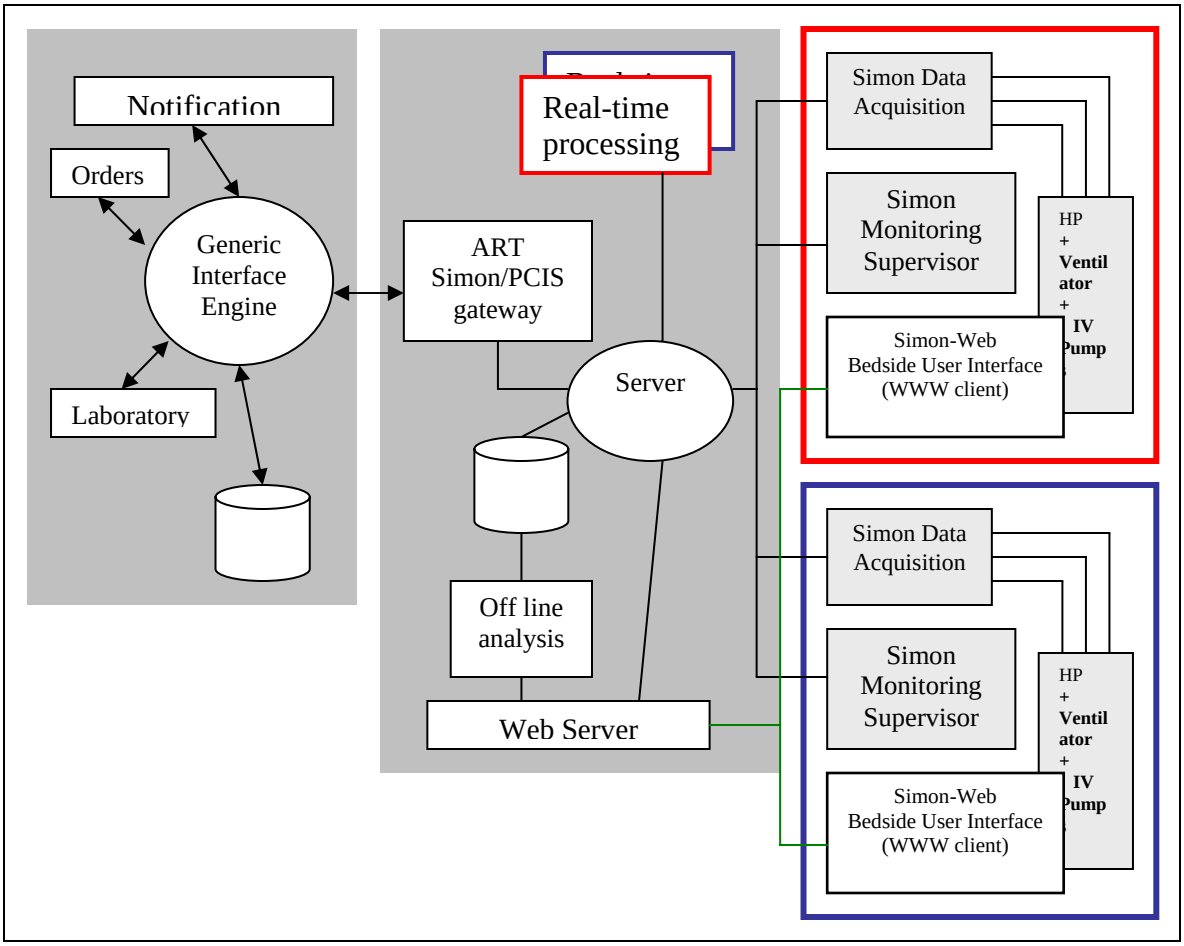

Figure 3: The architecture of the SIMON system as fielded in the CCU [10]

Revised SIMON Architecture

Design Constraints

The following design constraints were specified for the revised architecture:

Reliability – The system should be reliable enough to run 24 hours a day, 7 days a week with minimum human interference for non-life-critical decision support.

Expandability – Addition of new types of devices as well as extension of the system functionality (new alarming schemes, various types of data displays, etc.) should be simple.

Scalability – The system must be easily expanded to support additional beds and other intensive care units.

Flexibility – Installation of the system should not require more than a network connection at the bedside. Individual components should be able to be deployed on one or more physical machines, and reorganized as needed according to computational resource requirements.
Compatibility – While the core system components are designed to run on a particular platform (Windows NT), data interfaces should be available to external components running on a variety of platforms.

To meet these requirements, the new version of SIMON has been organized in three layers: the Data Layer, the Task Layer, and the Knowledge Layer.

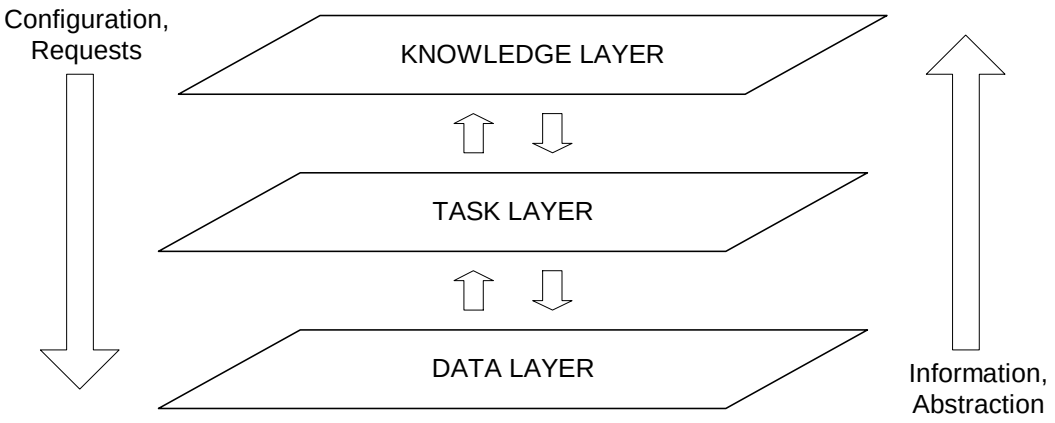

Figure 4: The Revised SIMON multi-layer reference architecture [17]

Data Layer

The main purpose for the Data Layer is to provide the information collecting system. Basically, the Data Layer acquires data from bedside devices and hospital databases and store data to storage. The Data Layer also distributes data to other components in other layers by providing a central point of access to all system data for message translation and message routing. The central component is the DataServer, which is the only component communicating directly with components in the Task Layer. There are also three main groups of components in

this layer: data collection, data storage and management, and web-based user interface. More details on these components are provided in chapter two.

Task Layer

The Task Layer acquires and monitors raw data streams from the Data Layer and detects user's configured events. When an event is detected, physicians, nurses etc. are notified.

Knowledge Layer

Components in the Knowledge Layer provide high-level reasoning capabilities for decision support, and configure the task layer to deliver notifications about events required for such processing. The Knowledge Layer is not discussed in this thesis (see [17, 18] for more details about the Knowledge Layer).

Componentization can significantly enhance the system in many aspects. Each component can be developed and tested separately, so new functionality can be tested and added with minimal effect on the whole system. The system can be scaled up by adding additional modules. Also, multiple and redundant components can be used to reduce the impact of failure of individual component. The system can be distributed on multiple processors and/or platforms which allows work to be shared. It also facilitates the management of the overall system.

This chapter has presented the requirements for an intelligent patient monitoring system in general; the motivation for this work has been presented as well as some lessons learned from our past experience. Finally, the revised architecture has been introduced. The following chapters provide more information about the various components of this system. The second chapter is a complete description of the Data Layer. It discusses the design, basic architecture, purpose and implementation of all modules in the layer. The third chapter presents the initial Task Layer and its modules. It explains the requirement for the Task Layer, its current architecture, and its implementation. Chapter four presents the improved Task Layer. This chapter describes a new architecture, the technology used, and its implementation. Chapter five concludes the thesis and discusses our experience with the current version of the system fielded in the trauma unit at VUMC, its results and evaluation, and future recommendations.

CHAPTER II

DATA LAYER

The Data Layer, as introduced in chapter one, is responsible for acquiring, organizing, archiving and distributing data for the entire system. Mainly, the Data Layer must be able to get raw data from both bedside devices and the hospital information system, store it persistently for future analysis, and organize it in order to distribute them to other layers. To achieve this, the Data Layer of the SIMON architecture has been divided into several interrelated components.

This chapter describes the various tasks performed by the Data Layer. Next, design options that have been taken are discussed and detailed. Finally, each of Data Layer's component is presented and discussed.

Functionality in the Data Layer

Data Acquisition

The Data Layer provides methods to acquire raw data from any required source. The raw data for the SIMON system can be generally classified into bedside devices data, and hospital information system data.

Data organization

Data organization is essential because of the vast amount of data acquired continuously for monitoring purposes. The Data Layer has to be able to efficiently manage the flow of data and pass the relevant data items to the other system components in the same layer and the Task Layer.

Data Archiving

Even though patient monitoring and alarming systems operate mainly on real-time data, historical data can be useful for other purposes such as medical research or the development of advice giving systems. Because the Data Layer is responsible for the acquisition and distribution of the data, its functionality also includes the ability to store these data to persistent storage.

Distributing Data

In the revised SIMON architecture mentioned before, the Data Layer is not responsible for processing the data. Hence, the Data Layer has to be able to distribute data to the Task Layer for data processing. In other words, the entire Data Layer acts as a server distributing data to the Task Layer.

Design Decision

Distribution

The Data Layer's architecture is based on distributing tasks to individual components. The Data Layer includes the following components: the DataServer, the Data Collection modules, the Web-based User Interfaces and the Data Storage and Management modules. The functionality of each of these components is detailed later in this chapter.

Communication

Most of the data handled by the Data Layer is acquired in real-time from bedside monitoring systems (some additional data can be originated in various databases of the hospital information system). To do this, the data is acquired in real-time by the Data Collection modules directly from the bedside devices and passed to the DataServer which stores and distributes these data appropriately. Because of reliability, performance, and wide acceptance, TCP/IP Socket [19] has been chosen to handle communication between the Data Collection modules and the DataServer. More details about the DataServer protocol can be found in [48].

Implementation

To facilitate code reusability, an Object-Oriented Programming approach has been chosen to implement the various components of the system. C++ existing libraries such as Microsoft Foundation Classes (MFC) [20, 21], Standard Template Library (STL) [22], and win32 API [23] have been used to code those components that need to operate in real-time. Java [24] has been used to implement the Web-Based User Interfaces components. Moreover, PERL [25] and JavaScript [26] have been used to implement other components.

Components

1. DataServer

The DataServer has been developed to act as a message translating and routing module for the data layer. The DataServer receives data from all data collection components and distributes the data to the Data Storage and Management components, as well as to components in other layers.

2. Data Collection components

The Data Collection components are responsible for retrieving data from the bedside monitoring devices or from a hospital database and for sending these data to the DataServer. In general, there is one Data Collection component per bedside device.

3. Web-based User Interfaces

Components in this group are those that deal with Web-based User Interfaces. Some components are used to create image files to be displayed on web pages while others are Java Applets that are embedded into various web pages.

4. Data Storage and Management component

The task of archiving data is performed by these components. These components are clients of the DataServer. Some of these components get the data and write them in normal text-file format to persistent storage while others populate databases, which can be easily retrieved later via SQL [27] commands.

DataServer

The DataServer is the key to data distribution since it is the central component in the Data Layer and it is responsible for routing data and message to and from every component in both the Data Layer and the Task Layer. Every data item acquired by the system is sent to the DataServer which then distributes it to the other system components.

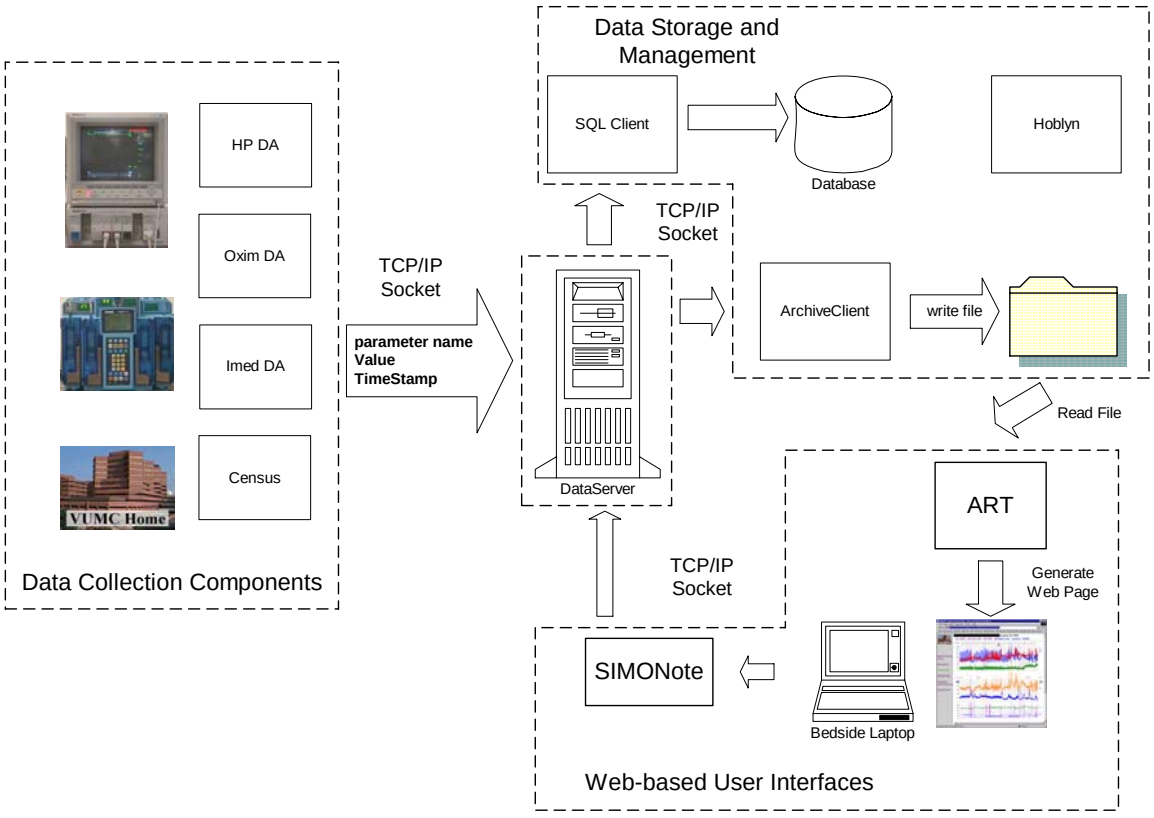

Figure 5: The Data Layer and various components (each of there components are describes in this chapter)

Implementation

C++ has been used to implement the DataServer. This decision was made because the DataServer has to deal with real-time data and performance is an important issue. Second, implementing a server application is a challenging task and the Visual C++ IDE by Microsoft provides many libraries needed to develop such applications. These include data structures (STL vector, STL maps, CPtrList, etc.) or well-built string classes (CString or STL strings) that facilitate and expedite the task of the developer.

Multi-Threading

Because the DataServer needs to be able to interact with several components simultaneously, it is multi-threaded. A main thread creates a new child thread and new socket to handle a communication every time a new connection from another data source or client is

established. All the threads communicate with each other via windows messages and synchronized global variables. A number of new messages have also been defined for this application in addition to the built-in windows messages. Multi-threading also improves reliability of the DataServer because failure of one component affects only an associated child thread in the DataServer while other child threads remain able to continue communicating with their component normally.

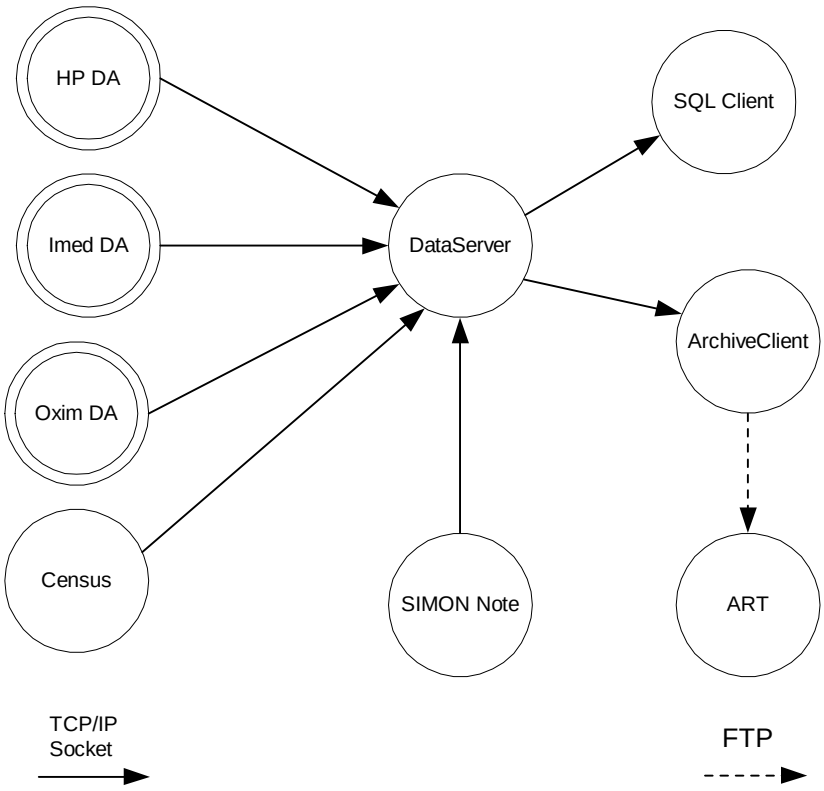

Figure 6: The Data Layer data flow diagram

DataServer Communication

The components communicating with the DataServer can be grouped into two categories. The first one is a group of Data Collection components. The Data Collection components acquire raw data from bedside medical devices and the hospital information systems. The bedside data are acquired and sent to the DataServer all the time whenever a patient is in a bed, and the data from the hospital databases are acquired periodically when needed. All the Data Collection

components send data to the DataServer via TCP/IP Socket. Currently, this is only a one-way communication because the Data Collection components only send messages to the DataServer. The second group of components is a group of client components. The components in this group include some components in the Data Storage and Management group and some components in the Task Layer. The client components also communicate with the DataServer via TCP/IP socket. However, both the DataServer and the clients can send and receive messages. The type of data that the DataServer sends to and receives from each client depends on each individual client.

Publisher-Subscriber Approach

When a connection between a new client and the DataServer has been established, the DataServer sends the list of data it can supply. To get the data from the DataServer, a client has to subscribe to the desired data streams. After that, the data to which the client subscribes are sent automatically every time the DataServer gets new data from the data collection components. Also, clients can unsubscribe if they do not require certain data any longer. When the DataServer receives an unsubscribe message from one of its clients, it stops sending these data items to that client. This approach is called Publisher-Subscriber or Observer pattern [42].

The publisher is the DataServer, which publishes data to the clients subscribers. This approach is better than a traditional request-reply approach, which is used in many client-server applications, because clients are not required to send message to the DataServer every time they need the data. However, to make this possible, the DataServer has to keep the information it needs to send the correct information to each individual client.

Dynamic Parameter List Updating

Because parameters from the Data Collection components may change over time (for instance, monitored variables can be added or removed), the DataServer must be able to keep the list of available data up to date. By checking the last receiving time of each data item, the DataServer can determine which data is valid. If the DataServer gets a new parameter from the Data Collection components or detects invalid data, the DataServer has to update its parameter list and notify every client.

Data Queue

Because of system load or network delays, it is possible that some clients may not be able to handle all data from the DataServer. To prevent loss of data, the DataServer should be able to queue the data for each individual client. As mentioned above, all the threads in the DataServer communicate with each other using windows message queues. Because each thread has its own message queue, when the main thread in the DataServer passes messages to client threads, the data and other messages are automatically queued.

Data Collection Components

The Data Collection components acquire data both from various bedside medical devices and from hospital information systems such as the census. Then this information is transmitted to the DataServer via TCP/IP Socket. A (non-exhaustive) list of parameters the data collection components can acquire is shown below.

HP_CPP – HP Monitor (HP) Cerebral Perfusion Pressure
HP_HR – Heart Rate
VL_PEEP – Ventilator Post End Expiratory Pressure, via HP VueLink (VL) module
VL_CI – Baxter Cardiac Index
Imed1_A_RATE – IV Pump 1, channel A Rate
CEN_MRN – Medical Record Number from hospital census system
Oxim_SaO2 – Ohmeda Biox 3740 Pulse Oximeter (Oxim) Oxygen Saturation
USR_NOTE – Notes from SIMON-Note

For example, HP_CPP is the Cerebral Perfusion Pressure acquired from an HP Monitor. The complete SIMON parameter list is in appendix A.

There is one Data Collection component associated with each bedside device. Other Data Collection components are used for querying periodically various databases of the hospital information systems.

Implementation

The data from bedside devices is acquired in real-time. For this reason, the Data Collection components that connect to bedside monitoring devices have been implemented in C++ for maximum performance. Another reason to use C++ is the availability of vendor-

provided libraries written in C which permit device interfacing (e.g., Hewlett-Packard's MECIF libraries [28]). Data from the hospital information system are only acquired periodically and other programming languages can be used for this purpose. In the current implementation of he system, PERL is used to implement the modules that gather census data.

Data Source Distribution

As mentioned earlier, there is one Data Collection component associated with each bedside device. Other Data Collection components include the Census component that gathers information about bed occupancy from the hospital databases. This information is used to associate a bed and the data acquired from the patient monitoring devices connected to this bed to a patient. Every data item acquired from bedside devices is labeled with a bed ID that allows the DataServer to identify its source.

Data Source Communication

All bedside Data Collection components connect to bedside devices via RS-232 serial interfaces using an RS-232 terminal server [29], a device that can be mounted near the bedside device. A terminal server acts as an Ethernet bridge and allows a remote computer to transparently access up to eight RS-232 serial ports via an Ethernet network.
For other components, different approaches are used. The Lab Data component acquires hospital laboratory information via an encrypted TCP/IP Socket. The Census component obtains bed information via periodic FTP file transfers.

HP Monitor Data Acquisition

This Data Collection component, called HPDA, acquires data from the HP monitors via RS-232 serial port. When one HPDA is started, it sends the list of parameters it wants to the monitor (currently, this list of parameters is hard coded in the HPDA modules). Once the communication is established, the HP monitor keeps sending data to the HPDA continuously [28]. In the current implementation of the system, HPDAs receive data from the monitors once a second. Moreover, by using Hewlett-Packard "VueLink" interfaces [30, 31, 32], HPDAs can get data from additional devices such as Servo ventilators [33] via the HP monitor. In the trauma

unit in which the system has been fielded, each bed is equipped with one HP monitor and there is one HPDA per bedside monitor.

IV Pump Data Acquisition

Like the HPDA, this Data Collection component, called ImedDA, acquires data from IV (intravenous) Infusing Pump via RS –232 serial ports. IV pumps can have one, two, and four infusing channels depending on the type of the pump. ImedDA detects the number of infusing channels and starts acquiring data. Unlike HPDA, ImedDA needs to send a request command to the IV pumps every time it needs data. ImedDA can get data from the pumps every two seconds (See protocol details in [34]).

Oximeter Data Acquisition

This Data Collection component, OximDA, is similar to the HPDA in that both are acquiring data via RS-232 serial ports and in both cases the bedside device sends the data periodically once the communication is established. The bedside pulse oximeter device currently used in the trauma unit (Ohmeda Biox 3740) can send data through its RS-232 port every two seconds [35].

Census

The Census component acquires information about bed occupancy. Monitored data can be associated with individual patient by using this information. Owing to patient safety and confidentiality concerns, this method is used to protect patient identity; therefore only authorized personnel can identify patients in historical data.

Web-based User Interfaces

Since current web technology provides a means of rapidly developing distributed clients, the World Wide Web is used to provide user interfaces to graphs of selected parameters. However, direct access to SIMON DataServer is resource intensive in term of hypertext markup language (html) formatting and implementing http protocols.

ART was created to handle the above tasks by graphing data and provide these graphs to a web server. Another web-based component of the system is the SIMON-Note applet that can be used at the bedside to annotate data and enter nursing notes.

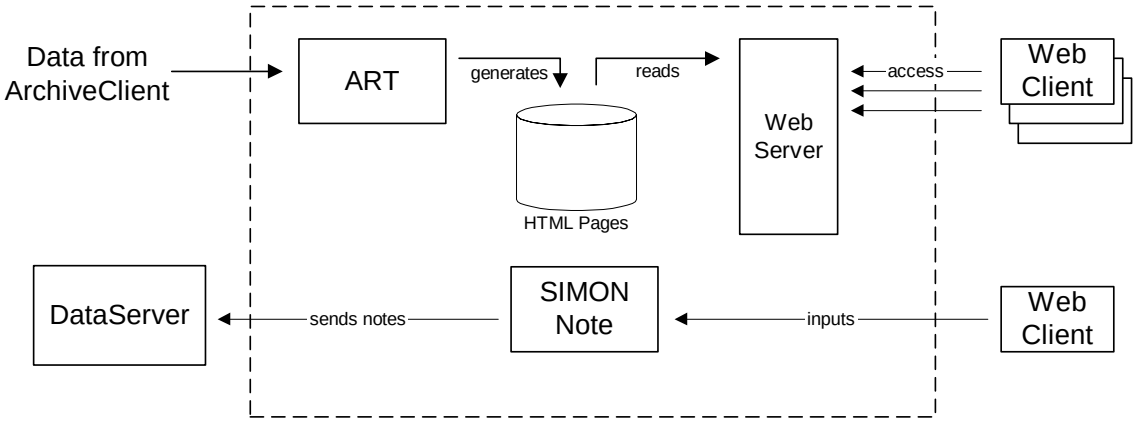

Figure 7: Web-based User Interfaces architecture with DataServer and Web Client

Implementation

Originally, ART had been implemented in C under OS/2. This program was then ported to Microsoft Windows NT to run with the new SIMON implementation. The SIMON-Note applet has been written in Java

ART

ART is a legacy component from our previous SIMON implementation that graph data and provide image files to a web server via a Networked File System (NFS) link. ART accesses data by reading text files generated by the ArchiveClient component of SIMON (see below) instead of interfacing directly to the DataServer. ART provides current views of patient monitor data, at various levels of resolution: 24 hours or one hour per screen. The ART graph showing in Simon-Trauma web site is shown below. The upper plots show cardiovascular data, including heart rate (HR), Diastolic arterial pressure (artBP), Mean arterial pressure (MAP), Mean pulmonary (PAP), and Non-invasive pressure (manBP). The middle plots show respiratory data, including pulse oximetry (02%Sat), cerebral perfusion pressure (CPP), and intracranial pressure (ICP). The lower bar plots show IV drug infusion rates, with the bar thickness corresponding to normalized dosage.

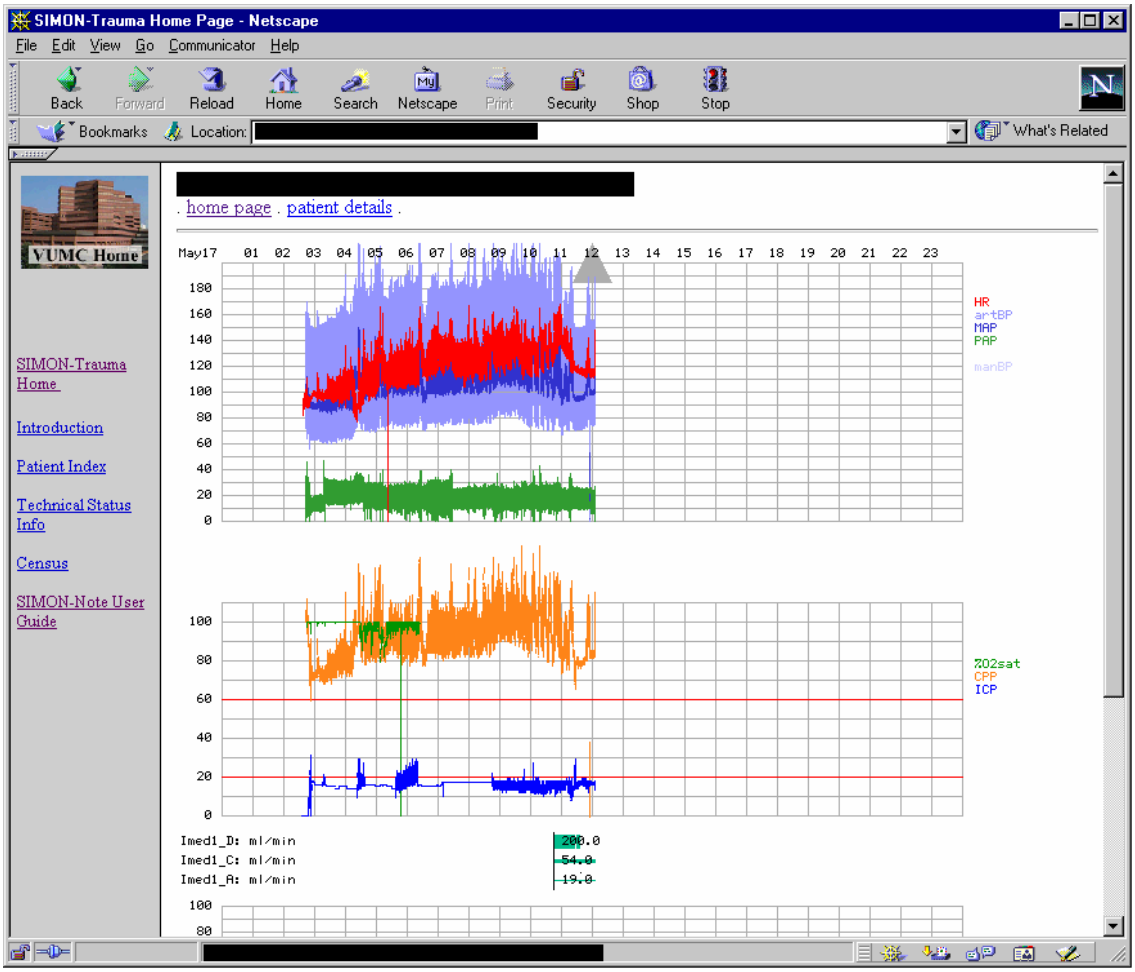

Figure 8: Simon-Trauma web site showing ART graph plotted over 24 hours.

SIMON-Note

SIMON-Note is an applet designed to facilitate the annotation of the data graphs at the bedside. SIMON-Note is a modification of a Java application developed by the Division of Biomedical Informatics in the Vanderbilt University Medical Center. It has been used by care providers to enter clinical notes. A right-click on the graph starts a new window that is initialized with the patient name, ID, and a time stamp. The users can then enter notes in free text format. SIMON-Note can also be used to create note templates that can be completed and filled in by the end user.

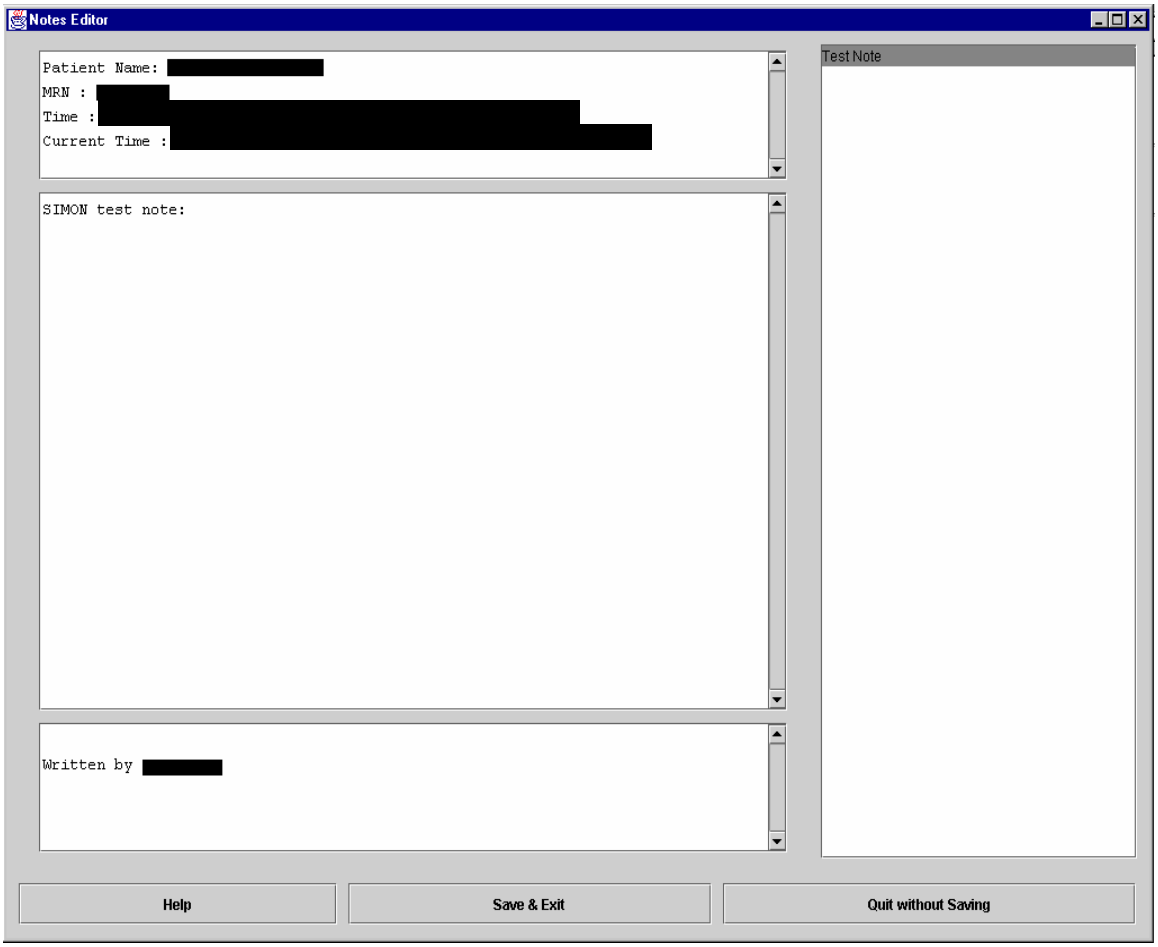

Figure 9: SIMON-Note user interface (Screen explanations are described in the next page)

Figure 9 shows the SIMON-Note user interface. The upper left text area of the applet is initially filled with the patient name, medical record number (MRN), and time information, while the user name is filled in the lower area. Care providers can enter notes in the middle area. The right area is a list of note templates. Care providers double-click an item in the list, which is defined in a configuration file on the server. The SIMON-Note applet uses JFC/SWING [36], a set of Java2 graphical user interface (GUI) components. Because Java2 is not fully supported by the web browser currently in use in the trauma unit (Netscape 4.7 and IE5), the Java2 Runtime Environment (Java Plug-in) from Sun Microsystems has been used instead of the original browser's JAVA virtual machine. Moreover, LiveConnect [37] is used to enable JavaScript and Java Applet to exchange data.

Data Storage and Management components

The components in this group are responsible for archiving and accessing data; they also implement the logic required to reliably associate data with patients.

The ArchiveClient and the SQLClient are components that provide data storage for access by other components. Another component is Hoblyn, which performs system and data management tasks.

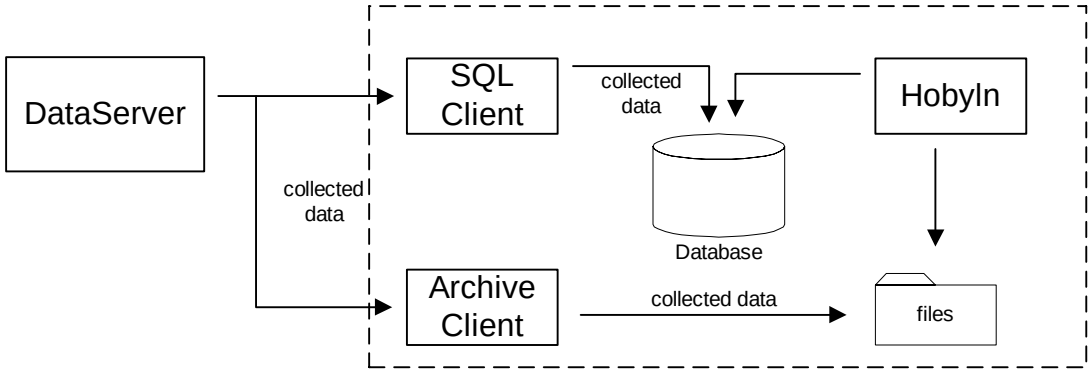

Figure 10: Data Storage and Management architecture with DataServer

Implementation

The ArchiveClient is implemented in C++ for performance reasons because it has to retrieve every data item passing through the DataServer and store these data in real-time. SQLClient is implemented with Microsoft Server 7.0. Finally, PERL is used to implement Hoblyn.

ArchiveClient

The ArchiveClient was designed to support legacy systems, such as ART and Hoblyn. The ArchiveClient initiates a connection with the DataServer by first requesting the list of available parameters and then subscribing to all the data. When it receives data from the DataServer, the ArchiveClient writes these data to text files. Data from each bed are put in different directory and each parameter in a separate text file. The DataServer dynamically manages the list of available parameters and automatically informs its clients when new data are

available or when data becomes unavailable. When the ArchiveClient is informed that a new variable becomes available, the ArchiveClient immediately subscribes to it. The rate at which data is written to file depends on the data source and every data item is stored with a time stamp. Table 1 shows a fragment of the file. The first column is the time stamp. The second is the value of heart rate associated with the time in the left column.

Table 1: ArchiveClient file

957979616 79.00
957979620 79.00
957979621 80.00
957979622 79.00
957979623 80.00

SQLClient

This component is akin to the ArchiveClient as it also subscribes to all parameters, but instead of writing data to text files as the ArchiveClient does, it populates a relational database implemented with Microsoft SQL Server 7.0. However, because the storage of every data point would increase the size of the database too rapidly, data points are only stored every 10 seconds. Also, a ten second median filter is applied to the raw data before they are down sampled. The SQLClient that reused some code from the ArchiveClient was implemented by Patrick Norris, a graduate student in the department of Biomedical Engineering, Vanderbilt University. More details about the SQLClient can be found in [17].

Hoblyn

Hoblyn was implemented in PERL by Patrick Norris. Hoblyn's task is data and files management. It maintains demographic information, archive, and move data files to and from directories as patient move in and out of monitored beds. This is an essential task that needs to be performed to guarantee that the association between data and patient is accurate at all time. Information about patient movement is obtained from the hospital admission-discharge-transfer (ADT) system. Hoblyn's task is complex because ADT information may lead or lag the actual time at which a patient is put into a bed and begins to be monitored by up to an hour or more.

Hoblyn has been fitted with a fair amount of logic that allows it to handle these situations robustly. Hoblyn also alerts project staff via e-mail and alphanumeric pager in the case of patient information ambiguity. Moreover, Hoblyn performs other tasks, such as verifying integrity of the bedside device data network and the NFS link to the web server. More details about Hoblyn can be found in [17].

CHAPTER III

CURRENT TASK LAYER

In the SIMON architecture, the Task Layer is responsible for detecting events in the raw data streams feeding the Data Layer. In particular, one or more components in the Task Layer connect to the DataServer and subscribe to required parameters. Once this is done, components in the Task Layer continuously monitor the data and notify care providers when predefined events, i.e. alarms, occur. Outputs from the Task Layer are also used by the Knowledge Layer for higher-level decision support tasks.

As in the Data Layer, a central point of interface for configuration of tasks and subscription to task outputs at the Task Layer level is a component called TaskServer. Generally, users supply information to the TaskServer about events that need to be detected and notification methods.

This chapter describes the design and implementation of the task layer and details its operation.

Introduction

Currently, the Task Layer has one component called the Alarm Detecting Client (ADC). In the current architecture, the ADC acts as the TaskServer. Users can supply information to the ADC via a configuration text file to specify events to be detected, and notification information (currently the email address of the care providers and the time of day).

As did the ArchiveClient in the Data Layer, the ADC communicates with the DataServer via TCP/IP Socket, but, unlike the ArchiveClient, it subscribes only to required parameters—i.e., parameters it needs to monitor.

From the DataServer point of view, the ADC is just a normal client. Several ADCs can thus run on several machines. This permits the configuration of many different events adapted to the needs of various end users. Each end user can define his/her own events and notification policy and implement these on any machine that is allowed to connect to the DataServer.

Workings of the ADC

The ADC is designed to be a small and simple program that monitors a number of parameters and detects predefined events. In the current implementation of the system, it reads a configuration text file when it is started. Hence, if the definition of events needs to be modified or adapted, the ADC must be restarted. A fragment of a configuration text file is shown below.

```
DS_IP 129.59.99.240
Alarm HR < 90 10
Alarm SvO2 < 60 15
Alarm CI < 2.5 0
Alarm ICP > 25 15
Label SpO2 02%Sat
E-mail_all suwanmk@vuse.vanderbilt.edu
E-mail karlkim.suwanmongkol@vanderbilt.edu
E-mail karlkim@iname.com
MaxGap 300
NormalGap 120
CheckNoise 30
SendTime 6:00
```

In the file, there are several important configuration items for the ADC. DS_IP is followed by the ip address of the DataServer to which this ADC will connect. Required events, threshold condition, and temporal intervals are put after the Alarm fields. For example, Alarm HR < 90 10 configures the ADC to subscribe to the heart rate (HR) parameter of every patient and monitor the value of the heart rate to check if it falls below 90 for 10 minutes for any of the patients. The E-mail field contains the e-mail address of the care provider who wants to be notified if this event happens. Here, notification is not done in real-time, but a report is sent once a day with a list of detected events. The SendTime field contains the time at which the e-mail message should be sent (6:00 means 6 am). The MaxGap and CheckNoise are fields used to contain parameters needed by the event detection algorithms. For instance, MaxGap is the time after which the ADC will stop monitoring a parameter if the DataServer stops sending values for it. CheckNoise is used by the event detection algorithm to reject spurious data points. The event detection algorithm considers an event to be meaningful if the heart rate remains below 90 for duration at least equal to CheckNoise.

When the ADC starts, it reads all the required information from the configuration text file. Then it tries to subscribe to all the required parameters. If the DataServer cannot supply some parameters at the time the ADC starts, it puts them on its list of desired parameters. As soon as the DataServer receives these parameters, it sends them to the ADC that begins monitoring their values.

Alarms in the ADC are categorized as current alarms and past alarms. Current alarms are events that are occurring and are being monitored by the ADC, while past alarms are events that have occurred, and those are not currently detected. Current and past alarm information are kept in xxx_CurrentAlarmFile.d and xxx_PastAlarmFile.d respectively (where xxx represents a bed ID). In addition, the ADC also keeps a log on how long each bed has been monitored in a duration text file. The ADC keeps monitoring and updating current alarms, past alarms, and duration text files. When the current time matches the notification time specified by a user, the ADC puts all current alarms, past alarms and the duration information separated by the bed ID in emails. These are then sent to the required email addresses. When this is done, the ADC starts monitoring the same events until the notification time is reached again.

Both files contain event information, including parameter name, a threshold condition, and the beginning and end time of events. While events in a current alarm file might be updated every minutes, events in a past alarm file are static, but new event can be added to the past alarm file. A fragment of the current alarm file is shown next.

```
HR < 90.00 from 04/03  6:21 to 7:25
ICP > 25.00 from 04/03 7:01 to 7:25
.
.
```

For example, if the ADC is still detecting an HR event at 8:00, the end time in the current alarm file above must be changed from 7:25 to 8:00. However, when the HR event ends, the event information will be moved from the current alarm file into the past alarm file. At the sending time, all events in current alarm files are copied into the past alarm files. After that, the ADC sends an email with all the events in the past alarm files. Finally, the past alarm files are deleted and the ADC continues monitoring data for the next day.

Implementation

ADC is a single executable written in C++. Several MFC classes and STL are also used here. The important classes are a CADCManager, a CADCFileManager, and an AlarmObject. Figure 11 shows a class diagram of these classes. The CADCManager class is a main instance that reads configuration text files, establishes the communication with the data layer, and monitors data streams. The CADCFileManager class is responsible for reading and updating alarm files. The ADC keeps information about individual parameters in an AlarmObject instance, and every time the ADC receives data from the data layer, it updates the state (m_ObjectState) of the instance. The way alarms are generated is discussed in detail in the next section.

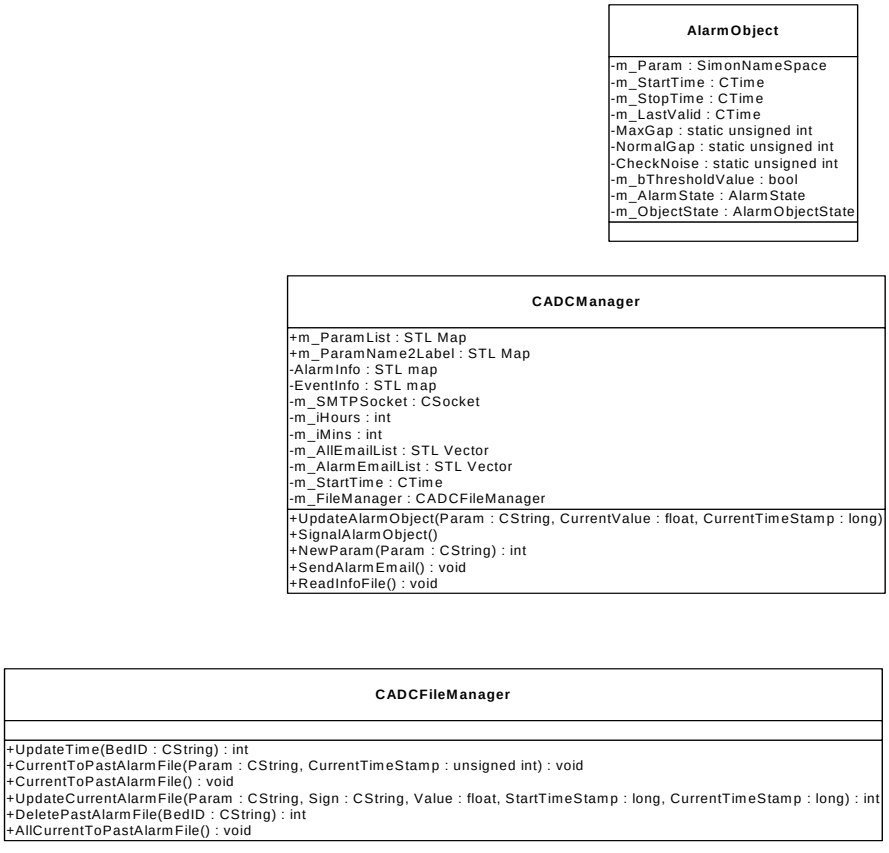

Figure 11: Class diagrams of important classes in the ADC

Generating alarms

As discussed previously, the ADC continuously monitors parameter values and generates alarms and stores those into text files whenever the pattern of data matches the event description. For example, the line **HR < 90 10** in the configuration text file instructs the ADC to compare every heart rate data point from every patient with a threshold value of 90. If the data is below 90 for a continuous interval of 10 min, an alarm is raised. But the data from the data layer may not arrive at constant intervals because of data dropouts due to system load, delays due to network traffic, or other minor glitches in the system. These gaps and data glitches must be considered because they can confuse the ADC, resulting in false alarms or missed events. Figures 12 and 13 show possible scenarios, including data gaps, and the logic that has been used to handle these situations.

Figure 14 shows the state diagram of the AlarmObject that explains how alarms are detected and written to the current and past alarm files. ThresholdValue is a threshold condition for each AlarmObject instance. The LastValid variable represents the last time at which the ADC received a data value for this parameter from the DataLayer. The Current system time is stored in the Now variable. The StartTime and StopTime variables are used to keep time information for the AlarmObject instance.

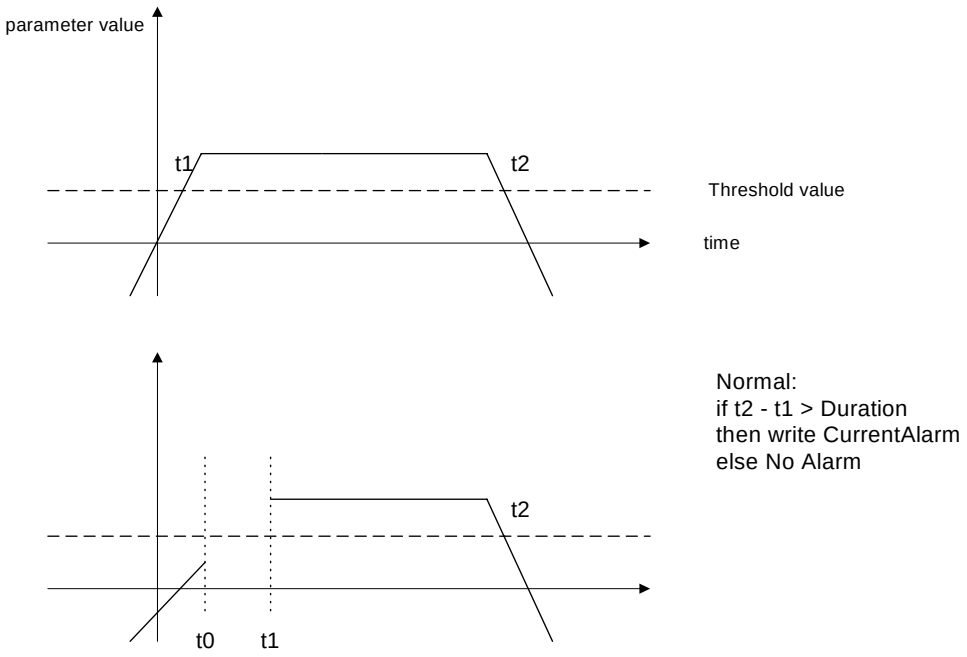

Figure 12: Possible sequence of data

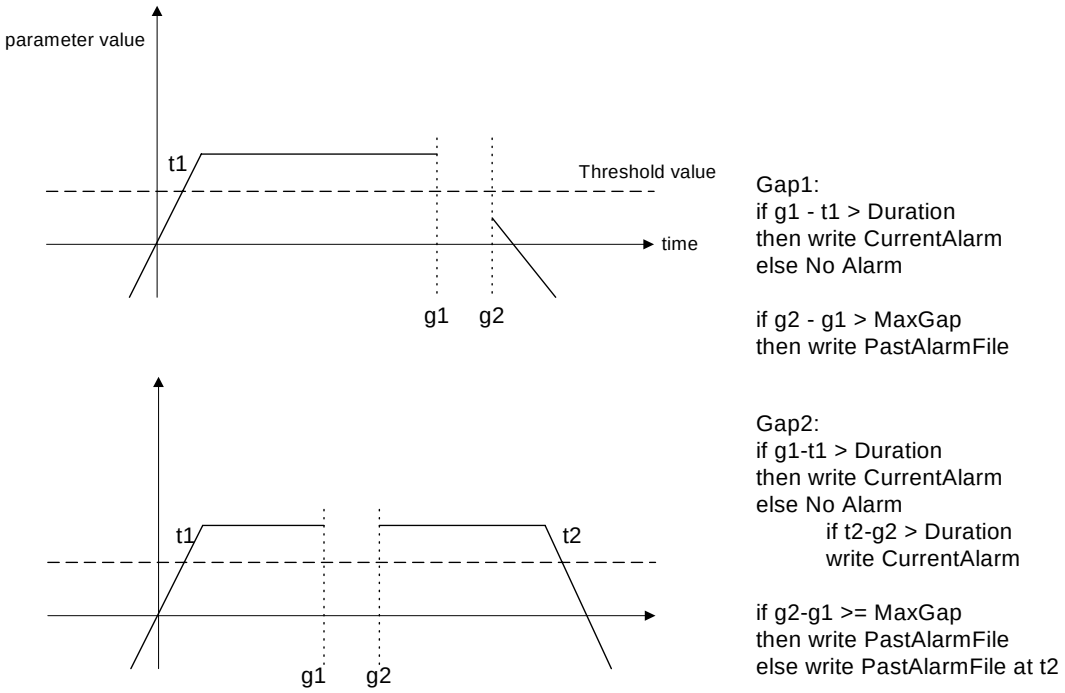

Figure 13: Discontinuous sequence of data

There are ten possible states for each AlarmObject. An initial state is a Monitor state. An AlarmObject instance will be in a Detected_Threshold state whenever the ThresholdValue is true longer than the CheckNoise time. Whenever there is a gap (now – LastValid > NormalGap) in a stream of data, an AlarmObject instance will change to either one of the three Gap states. To which state it will change depends on its current state. If the length of the gap is longer than MaxGap, from any Gap state, an AlarmObject instance will change to the DeleteAlarmObject state. In this state the instance will eventually be removed from the ADC. Those above six states excluding DeletedAlarmObject state can be grouped into one large Monitor state. The other three states which can be grouped into one large Alarm state are Detected_Alarm state, Gap_alarm, and Detected_Alarm_Wait state. The CADCManager checks all AlarmObject instances in the Detected_Alarm state and updates the corresponding current alarm files every minute. The current alarm files are also updated every time there are transitions among these three states. Past alarm files are updated whenever an AlarmObject instances' state changes to the Monitor and Delete states.

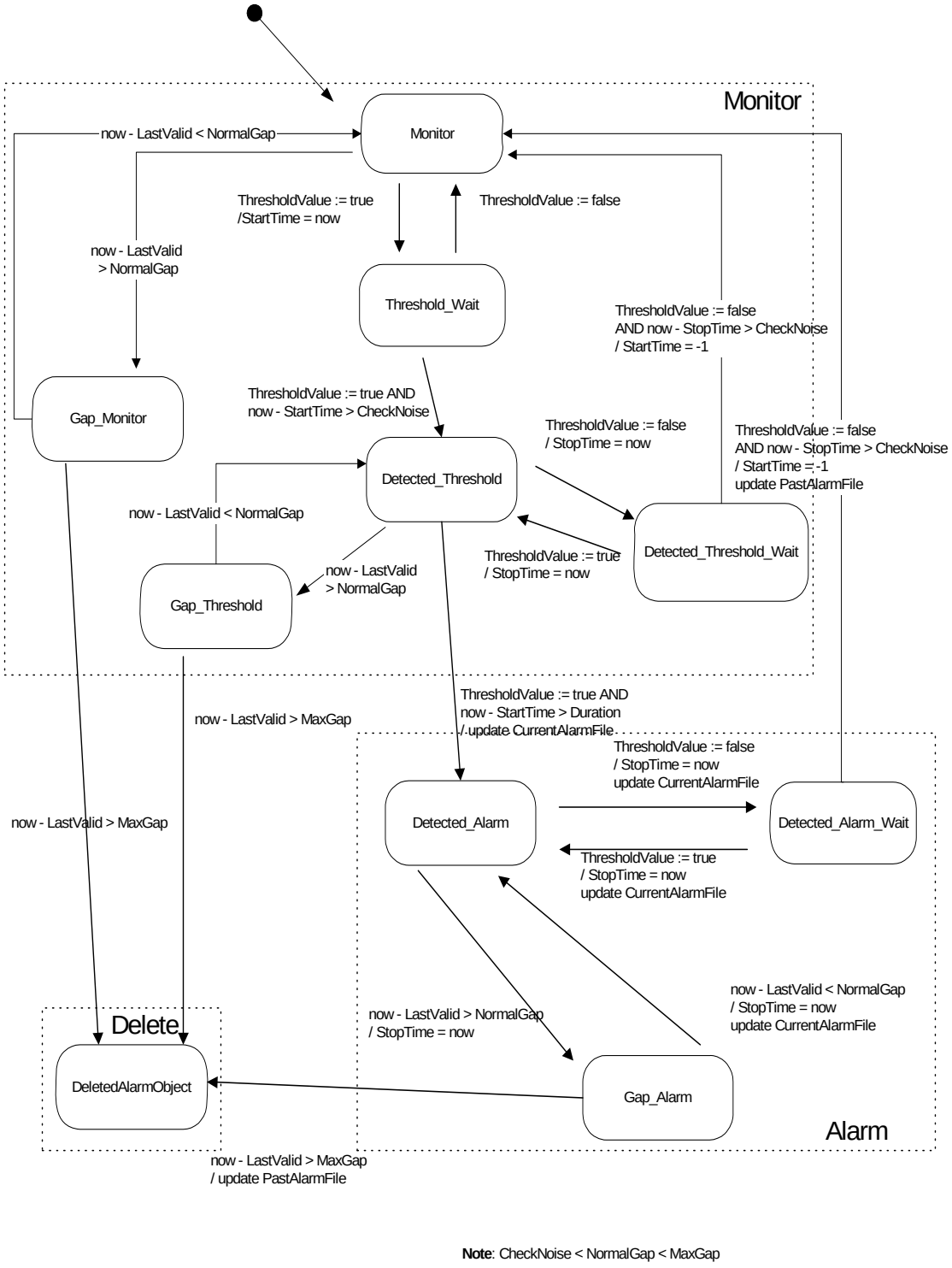

Figure 14: State diagram that shows how alarms are generated

CHAPTER IV

REVISED TASK LAYER

The Task Layer described in the previous chapter, with one ADC component, was deployed in the Trauma Unit at Vanderbilt University Medical Center. The ADC has been reliable and has detected many alarms since it has been fielded (see chapter five for more details). However, there are several issues that were not fully addressed in the initial development of the ADC. These issues are discussed below by evaluating the current Task Layer.

Capability

The ADC described earlier is able to detect only threshold condition alarms from a single parameter. However, in many situations, important alarms involve events detected from more than one parameter and also involve more than simple threshold conditions. The complexity of this type of events depends on the number of parameters to be monitored, the event definition for each parameter, and the way in which these single-channel events need to be combined to define a clinically significant alarm. In addition to event detection, the other equally important matter is the way by which users have to be notified once an event has been detected. User notification methods can be real-time notifications— i.e., an alarm is sent whenever alarms are detected, and archive notifications—for future research purposes. As mentioned above, the ADC does not support real-time notification methods since it keeps all the alarms in text files and does not send them to users via email until setup time. To handle real-time alarms, the Task Layer should be able to use other notification methods such as paging beepers or updating screens via Graphical User Interfaces (GUIs).

Expandability

The ADC has been originally designed as single component software. In every software development cycle, applications have to be modified to add to or improve their functionality. After that, they must be redistributed to give users access to the new functionality. However, the redistribution process itself is not easy to perform in a real-time system because the running

ADC must be stopped before the modifications can be deployed. Suspending the current ADC potentially results in losing information and missed alarms.

In addition, modifying the applications is often error-prone since the process invariably introduces new bugs to the application. As such, a better design strategy is to separate monolithic applications into a group of distributed components with restricted interfaces, so only the modified components need to be redistributed and tested.

Flexibility

To monitor parameters 24 hours a day, the ADC has to run all the time. This is inconvenient if the ADC has to run on the user's machine because (s)he needs to keep it running all the time as well. Also, the only way to configure the ADC in the previously described implementation is to modify text files that are read when it starts. This has two disadvantages. First, the ADC cannot be reconfigured dynamically. Second, text files may not be the best configuration method for all users. Clearly, the task layer should provide a more flexible solution for its configuration and dynamic reconfiguration.

Performance

The ADC acts as a normal client to the DataServer. Multiple ADC instances running simultaneously force the DataServer to create more threads to handle each ADC (see chapter two). This may downgrade performance of the whole system substantially because monitored parameters may be redundant (i.e., more than one ADC may request the monitoring of the same parameter). Reducing this redundancy helps to retain the real-time performance of the entire system. Solving this problem can be done by creating a separate task that connects to the DataServer from other tasks, so if users configure redundant parameters, the DataServer needs to send the parameters only once. The whole task layer can thus have only one instance of the part connecting to the DataServer.

Design Decision and Architecture

Based on the observations and requirements discussed above, the revised Task Layer is divided into several components. The main functionality of these components is

1. Acquisition of data from the DataServer
1.2. User Interface
1.3. Events Monitoring and Alarm Detection

In addition, to cope with the complexity of notification methods and multiple-parameter events, the event monitoring part can be divided further into four levels.

1. Single-Source Event Level (SSE Level)
1.2. Multi-Source Event Level (MSE Level)
1.3. Notification Level
1.4. User Interface Client

The Single-Source Event Level works as a normal client of the DataServer (i.e., an instance of the Single-Source Event Level connects to the DataServer and retrieves data from the DataServer through a TCP/IP socket), so the single-parameter event detection algorithms presented in the last chapter and the code required to communicate with the DataSever can be integrated. The Multi-Source Event Level handles multiple-parameter events by monitoring all related notifications of single-parameter events from the Single-Source Event Level. The Notification Level works with multiple clients acting directly with the users. The users specify required events, recipients, and notification information to the notification level via the User Interface Client. All three levels run concurrently as long as there are events to be monitored. Users can define their own events with customized clients and close the client program or turn off their computer if they do not want to be notified via their application. Other alarm notification methods, such as email or paging beepers, are kept in the Notification Level, thus the users do not need to stay in front of their computer to be notified. Figure 15 shows the architecture of the new Task Layer called the AlarmServer

The first two levels are currently implemented in C++ as Component Object Model (COM) out-of-process servers. COM specifications which embody several successful programming concepts such as Object-Oriented Model, Client/Server Model, and Dynamic Linking Library, help to simplify the process of distributed application development. COM objects can be implemented in many programming language. Also, C++ with the Active

Template Library (ATL) has been selected to speed the development and eliminate unnecessary coding. More details about COM/DCOM and ATL can be found in [38, 39, 40, 41].

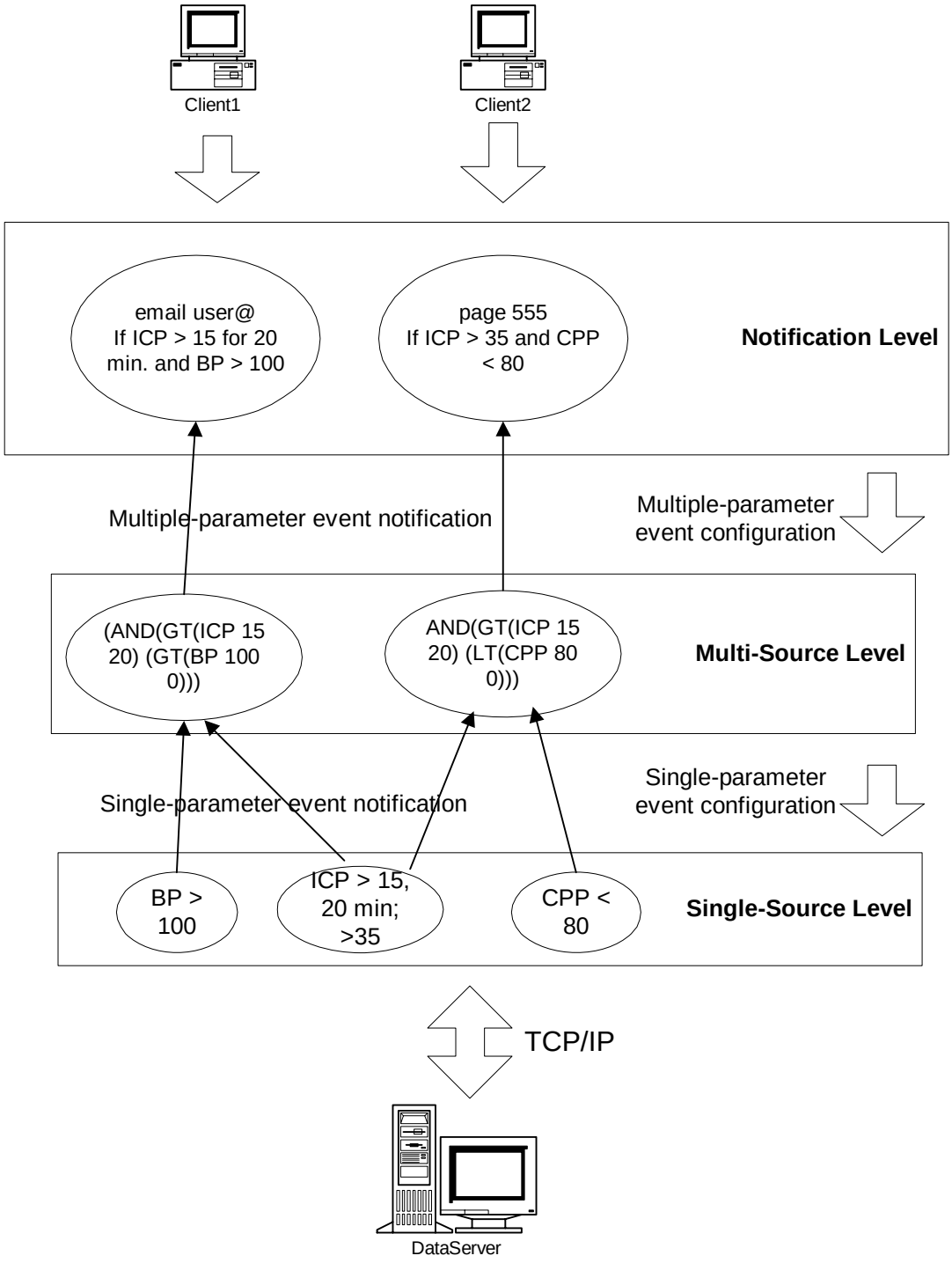

Figure 15: The Task Layer architecture with three separated levels and client applications

Single-Source Event Level (SSE Level)

The main functionality of the Single-Source Event Level is similar to the ADC discussed in the last chapter. The Single-Source Event Level communicates directly with the DataServer via TCP/IP Socket. In other words, the Single-Source Event Level subscribes to required parameters, receives data, and monitors the raw data.

Requirements

The set of classes in the Single-Source Event Level provides the following functionality.

- Connect to the DataServer via TCP/IP Socket
- Subscribe and unsubscribe to required parameters. If more than one client needs the same parameter, the Single-Source Event Level subscribes to the parameter only once.
- Acquire data from the DataServer in real-time.
- Provide an interface to the Multi-Source Event Level to enable the Multi-Source Event Level to setup required single-parameter events.
- Notify components when events occur through a call-back interface.

The design of the first ADC was rather inflexible. As a consequence, modifying or adding monitoring or processing algorithms was a demanding task. The design and implementation of the SSE Level were influenced strongly by the need to support modification of low-level detection algorithms without impacting other levels in the AlarmServer.

Implementation

Communication

Since the DataServer is able to communicate with other components through TCP/IP sockets, the Single-Source Event Level to DataServer communication is coded in C++ using Winsock2 API. Also, distributed objects method calls based on DCOM from Microsoft is the technique used to communicate with the Multi-Source Event Level. The Single-Source Event Level has public programming interfaces which DCOM clients can use to communicate.

DCOM

Components in the SSE Level reside in an out-of-process COM module so that the components can exist independently even when there is no client running. Another important issue is the development method. Graphical user interface is not a critical necessity at this level, and Microsoft provides speedy and simple ways to create COM objects with ATL. Therefore, ATL has been used to develop SSE Level COM objects instead of MFC COM and generic C++. The Standard Template Library (STL) is also used as a substitute for MFC collection classes.

Clients can communicate with the SSE Level only through a set of DCOM interfaces. The SSE Level has to be able to handle multiple clients at the same time, so the Multithreaded Apartment Model has been used. In addition, when events have been detected, the SSE Level notifies the client via COM interfaces as well. Connecting Objects techniques are used to provide this functionality.

Inside the Single-Source Event Level

The CSSEManager Class

Several classes are created to handle the aforementioned functionality at this level. The most important one is a CSSEManager. Components in the Multi-Source Event Level can communicate with the SSE Level through ISSEManager interfaces implemented by the CSSEManager. The set of interfaces used in the SSELevel is shown below.

```
interface ISSEManager : IUnknown
{
      [helpstring("method SetupThresholdEvent")] HRESULT
SetupThresholdEvent([in] ThresholdInfo SSEInfo, [in] EventObjID
SSE_ID);
      [helpstring("method DeleteEvent")] HRESULT DeleteEvent([in]
EventObjID SSE_ID);
};

interface ISSEProcess : IUnknown
{
      [helpstring("method ProcessParam")] HRESULT ProcessParam();
};

interface INotifySSE : IUnknown
{
      [helpstring("method NotifySSEState")] HRESULT NotifySSEState([in]
EventObjID SSE_ID, [in] EventState State);
      [helpstring("method CheckSSEObj")] HRESULT CheckSSEObj([in]
EventObjID SSE_ID, [out] int* Check);
};
```

The three IDL DCOM interfaces used in the SSE Level. Both ISSEManager and ISSEProcess are implemented in CSSEManager. INotifySSE which is an event sink interface is actually implemented in the CMSEManager (in MSE Level). The CSSEManager is a singleton meaning that only one instance of the CSSEManager is created, so there is only one point of access to the SSE Level (See Singleton design pattern [42]). A unique instance of the CSSEManager is first instantiated by COM when the MSE Level module starts. Components in the MSE Level call the **SetupThresholdEvent()** method with ThresholdInfo (i.e., parameter name, logic type, threshold value, and duration) and a unique single-parameter event object is generated in the MSE Level. Then the CSSEManager creates an object if the object monitoring the single-parameter event does not exist.

By assigning the task of generating the object to the CSSEManager, the duplication of objects that monitor the same parameter event is avoided. Also the MSE Level needs to have only information about pre-approved types of single-parameter events. The way the object that monitors the event is created (i.e., type of object, multiple objects, or redundant object) is up to the CSSEManager. The MSE Level can call the **DeleteEvent()** method with a single-parameter event object id to delete the object as well.

Figure 16 shows the CSSEManager and its interfaces. The ISSEProcess interface is for interacting with the DataServer communication part. The ISSEProcess interface will be presented later.

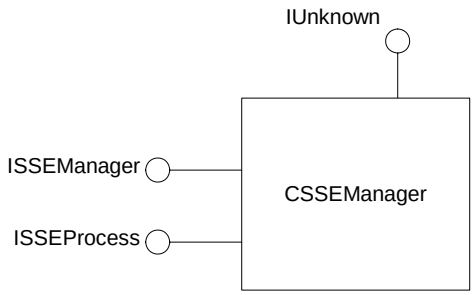

Figure 16: The CSSEManager and its interfaces

The SSEObj Class

In this architecture, all objects created by the CSSEManager are instances of a class inherited from the SSEObj abstract class. The base class SSEObj contains some fields and provides some methods and void virtual functions the subclass has to implement. A **Notify()** method is used to determine if components in the MSE Level should be notified. An **UpdateParam()** method updates the value of parameters for each single-parameter event object. A **CheckEvent()** method implemented only in subclasses is important because the behavior of a single-parameter event object is defined in this method. For example, GreaterObj is used to monitor a greater-than-threshold type of a single-parameter event.

Figure 17 shows the SSEObj class with two subclasses currently implemented. All three methods are called by the CSSEManager, and the CSSEManager does not need to know the type of a single-parameter event object it is calling. Part of the code used to detect threshold event in the ADC is also reused here (see figure 14 in chapter three).

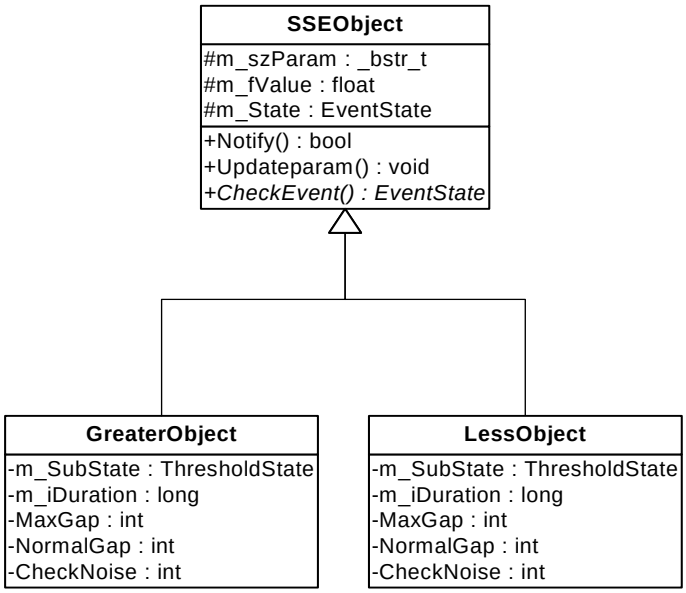

Figure 17: UML class diagram of the SSEObject, GreaterObject and LessObject with some important data members and member functions.

TCP/IP Socket Communication

Another important functionality of the SSE Level is to handle TCP/IP socket communication. With ATL, Winsock2 API is used here instead of the MFC CSocket that implements the communication part of many components in the Data layer. Avoiding MFC classes completely in this component removes the overhead of the large MFC runtime DLL. This part was designed to run in other windows threads (i.e., thread that implements windows message queue (for more details about windows and Winsock programming, see [23, 43, 44]).

As mentioned in chapter two, the DataServer sends large amount of data continuously to its client. As a consequence, the communication task should be ready to receive data, and update the value of its monitored variable all the time. When the thread gets raw data from the DataServer, it informs the CSSEManager through the ISSEProcess which has one method, **ProcessParam()**. Subsequently, the **ProcessParam()** method in the CSSEManager calls the three methods of the SSEObj as explained above.

Multi-Source Event Level (MSE Level)

The MSE Level receives information about multiple-parameter events from the Notification level and configures the single-parameter event in the SSE Level. The Notification Level sends an event description that is a Boolean combination of threshold condition in the form of a string to the MSE Level through a DCOM interface. The MSE Level then parses the string and creates a C++ object to monitor the multiple-parameter event which in turn properly creates a single-parameter event objects in the SSE Level. When the MSE Level gets notified by the SSE Level, the multiple-parameter event object evaluates its state and then notifies the Notification Level as necessary.

Requirements

While the basic requirements of the SSE Level are identical to the ADC, the requirements of the MSE Level are established with the necessity of providing a method to detect multiple-parameter event. The MSE Level provides the following functionality

- Provides a call-back interface for the SSE Level.

- Provides the interface for the Notification Level to configure multiple-parameter events.
- Notifies components in the Notification Level through a call-back interface

Implementation

While the SSE Level needs to be able to communicate with the DataServer through a TCP/IP socket, the MSE Level can use DCOM technique for its communication. The MSE Level is also implemented as an out-of-process COM module. As was the case for the SSE Level, for simplicity and because a GUI is not needed in this level, the ATL and STL have also been used to implement the MSE Level.

Inside the Multi-Source Event Level

The CMSEManager Class

As does the CSSEManager in the SSE Level, the MSE Level has a central class CMSEManager which implements two interfaces. The first interface is IMSEManager. This interface is very similar to ISSEManager since both provide two methods to add and delete an event at the lower level. The first method of the IMSEManager is **AddMSE()**. The Notification Level calls the method with MSEInfo (i.e., a Boolean expression in the form of a string) and a unique multiple-parameter event id. The Notification Level calls **DeleteMSE()** with the event id to delete the event. The two interfaces in the MSE Level are shown next.

```
interface IMSEManager : IUnknown
{
      [helpstring("method AddMSE")] HRESULT AddMSE([in] MSEInfo Info,
[in] EventObjID MSE_ID);
      [helpstring("method DeleteMSE")] HRESULT DeleteMSE([in]
EventObjID MSE_ID);
};

interface INotifyMSE : IUnknown
{
      [helpstring("method NotifyMSEState")] HRESULT NotifyMSEState([in]
EventObjID MSE_ID, [in] EventState State);
      [helpstring("method CheckMSEObj")] HRESULT CheckMSEObj([in]
EventObjID MSE_ID, [out] int* Check);
};
```

The INotifyMSE is implemented by the Notification Level and is called back by the CMSEManager when it wishes to give information about a multiple-parameter event. Moreover, the CMSEManager also implements INotifySSE (see inside the Single-Source Event Level) which is a call-back interface. The CSSEManager calls **NotifySSEState()** when it needs to notify the MSE Level about a single-parameter event.

MSEObj, ExprObj, BoolObj, and SSETypeObj

An object that the CMSEManager creates is an instance of the MSEObj. The instance parses the Boolean expression and creates a group of the BoolObj and the SSETypeObj objects. A group of the BoolObj and the SSETypeObj structure is a binary tree, in which every leaf in the tree is the SSETypeObj, and every BoolObj is an internal node. Both the BoolObj and the SSETypeObj are inherited from the ExprObj. Figure 18 shows the class diagram of these classes, and figure 15 shows some example of Boolean expression and their associated tree.

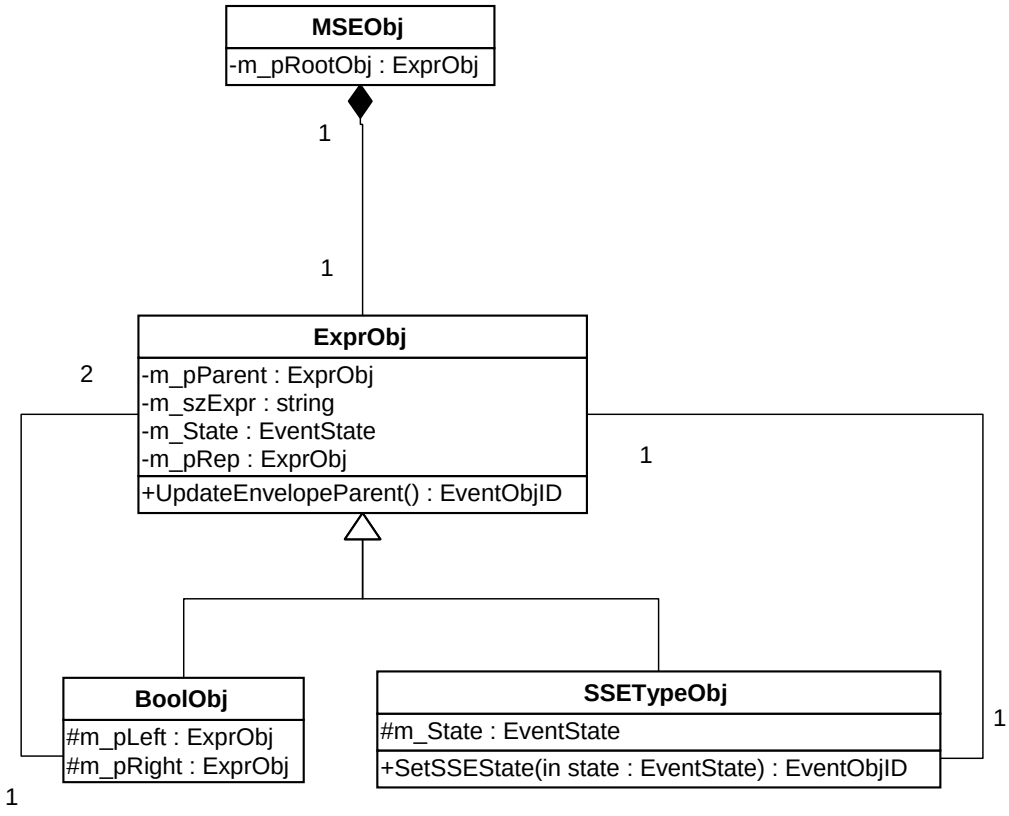

Figure 18: UML class diagram of the MSEObj, ExprObj, BoolObj, and SSETypeObj showing class structures and their relationships.

An envelope/letter class idiom is used to implement a virtual constructor technique. This is used to determine what type of object (i.e., BoolObj or SSETypeObj) will be constructed from the expression (see [45] for more details). The ExprObj is an envelope class while the BoolObj or the SSETypeObj is a letter class. MSEObj first creates the root of a tree that must be an instance of the BoolObj, then the root creates its children and the children parse the sub string. The process proceeds until all leaves which are instances of the SSETypeObj are created. Figure 19 also presents the tree and the sub string each node will parse. The sub string which also has the same format will be processed from top to bottom.

When an instance of the SSETypeObj is created, the MSE Level also notifies the SSE Level. Thus, if necessary, the SSE Level creates a single-parameter event object associated to the instance of the SSETypeObj.

Process of Notification

The notification process begins when any single-parameter event object detects a required event. When this happens, the CSSEManager informs the CMSEManager that a single-parameter event is detected, then the CMSEManager calls a method **SetSSEState()** on each instance of the SSETypeObj that needs to change its state. The **SetSSEState()** method updates each single-parameter event state and notifies its parent by calling **UpdateEnvelopeParent()** on its parent in a tree which is an internal node (i.e., an instance of BoolObj). The parent then evaluates its state. For example, a state of an instance of BoolObj having type “AND” will be “Alarming” if its children are both in the “Alarming” state. Moreover, if the state of the instance of BoolObj is changed, **UpdateEnvelopeParent()** of its parent will be called next and so on.

In other words, updating the state of a multiple-parameter event object starts from the bottoms with single-parameter event objects and progresses up to the root which is an instance of BoolObj. This bottom-up approach ensures that every object is updated properly and efficiently. Each object communicates with its neighbors in the hierarchy (tree) only. This approach is similar to an organization of decision processes in the Process Trellis Software Architecture [49].

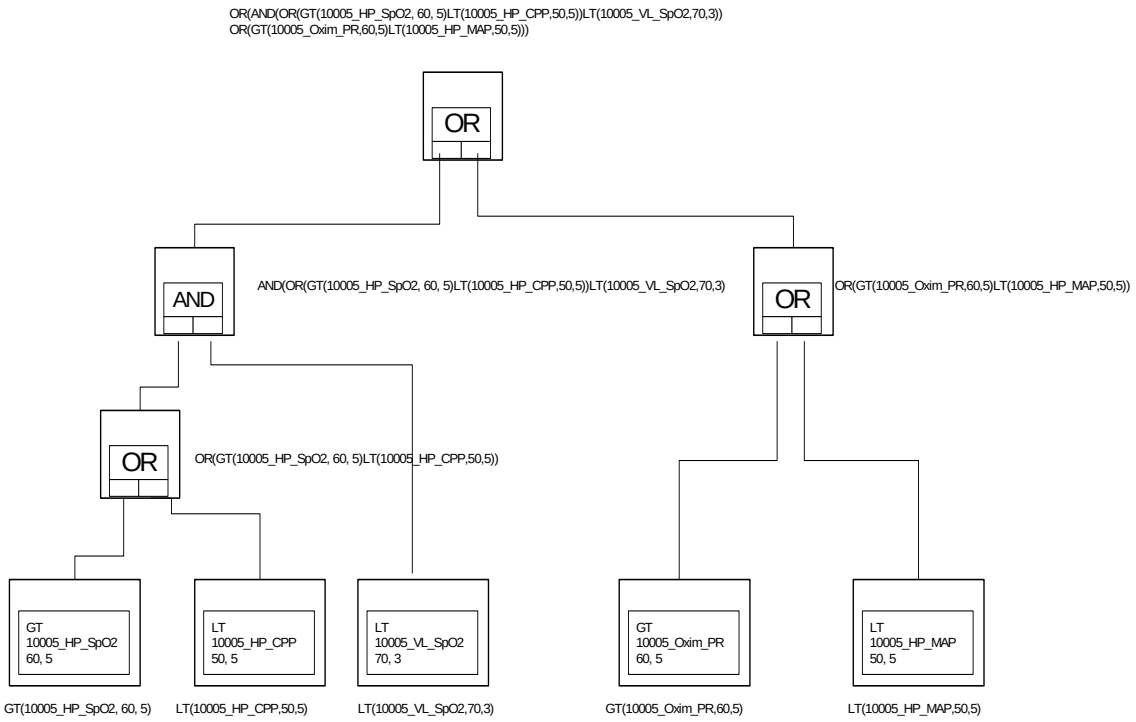

Figure 19: A sample of string and tree for multiple parameter event expression.

((10005_VL_SpO2 < 70 for 3 minutes)
^((10005_HP_SpO2 > 60 for 5 minutes)v(10005_HP_CPP < 50 for 5 minutes)))
v((10005_Oxim_Pr > 60 for 5 minutes)v(10005_HP_MAP < 50 for 5 minutes))

Notification Level

In the AlarmServer architecture, each lower level acts as a client of the upper level. The Notification Level, not yet fully implemented, is a client of the MSE Level. The Notification Level receives event configuration data from users via client GUI interfaces. The information consists of multiple-parameter events and associated notification methods. Because the Notification Level object resides in an out-of-process COM server, after users have configured an event detection task, they can close their client application.

Multiple-parameter event

The format of a multiple-parameter event sent from a client to the Notification Level is a Boolean combination of single-parameter events in a format similar to the one used to send from the Notification Level to the MSE Level.

Notification Method

The only notification method available from the ADC is to send email to users. Sending email to users, as mentioned before, is not real-time and not flexible. Other possible notification methods that could be implemented include:

- Notify the client GUI: this can be done in real-time and users can effectively disable this notification method by closing the client program.
- Notify the user via pagers: This method is similar to the first one because it is done in real-time. However, the advantage is that users can be notified everywhere via their pager.

CHAPTER V

CONCLUSIONS

Intelligent patient monitoring and management systems are complex and involve all aspects of information processing. These systems need to acquire physiological data as well as data from the hospital information system (HIS) under real-time constraints. In addition, these systems must provide efficient mechanisms to store, integrate, correlate, and provide a large amount of information to clinical users.

The earlier version of the SIMON system was designed to achieve most of these tasks and it was fielded in a real intensive care unit. Shortcomings of this architecture as well as the rapid evolution of both hardware and software technology lead to the redesign and reimplementation of this system.

The new version of the system is designed to provide a robust, expandable, and open architecture to support intelligent patient monitoring research. This is achieved by partitioning the system into multiple components and distributing data, tasks, and knowledge among these components.

Monitoring and interpretation tasks also play essential roles in many important applications. There are ongoing researches contributed for building tools and architecture to enable rapid construction of such system, such as MAITA [50]. The SIMON system is different from the MAITA system because it is designed to provide an open architecture that can be fielded in a variety of critical care units and configured for a wide range of monitoring applications. The MAITA system, however, provides more general architecture, tools, and processes for constructing knowledge-based monitoring system. The MAITA system architecture is divided into four parts consist of Monitor Processes, Knowledge Bases, Edit/Query Tools, and Templates & Models while the SIMON system is divided into three layers. The Monitor Processes in the MAITA system can be compared to the Data Layer and the Task Layer as it acquires data from data sources, processes these inputs, and reports the results in accordance with an alerting and display model. The Knowledge Bases and Edit/Query Tools parts also have a concept similar to the Knowledge Layer in the SIMON system.

This chapter begins with the evaluation of both the Data Layer and the Task Layer. Next, the current status and results are presented. Finally, future research directions and recommendations are discussed.

Data Layer

Overall, the SIMON Data Layer provides effective distribution of data, and the architecture supports easy addition of data sources and clients. New individual components can be straightforwardly tested and added to support additional tasks by developing new components that follow the SIMON Data Layer protocol and name format. Also, existing components can be easily modified without stopping one or whole of the Data Layer's components. For example, if a new data source is required, the new data source acquisition module can be developed and dynamically added to the system without interfering with the current system, i.e. the DataServer and other components do not required to stop running. This is also true when a new client is added to increase the functionality of the Data Layer. Using TCP/IP Socket based-interfaces makes it easy to develop new component communicating with the DataServer because TCP/IP Socket based-interface is available in many computing platform and programming languages. In the current implementation of the system, the DataServer written in C++ can communicate with the Census component implemented in PERL and SIMON-Note implemented in Java. TCP/IP Socket also permits the easy distribution of components onto several machines. More details about the protocol can be found in [48].

Also, if the new components are written in C++, many classes can be reused in order to decrease development time. For instance, communication and data structure classes, found in the DataServer, Data Collection and client modules can be reused.

The centralized data server provides convenient, open access to a large amount of data. By making the DataServer a central point of access for the data, all the raw data sent from the Data Collection components to clients have to pass through the DataServer. This permits simple data organization and management scheme since the DataServer is the only component that has to be modified. For example, the DataServer was enhanced by adding noise filtering and parameter name filtering to remove invalid incoming data before sending it out to clients.

In terms of system scalability, the main limiting factor for the number of components that can be added to the system is the available hardware resources. In addition, the performance

of the whole system, i.e., how fast the system can handle the data, depends mostly on the number of Data Collection components running simultaneously on a system. There are two issues that need to be considered. The first issue is that these components, especially components acquiring data via RS-232 serial ports, do consume a fair amount of hardware resources. When several beds are monitored simultaneously, which increases the number of active Data Collection components, data dropout can occur because the components are not able to process data fast enough. This performance issues can be partially solved by distributing other resource consuming components, such as ART or some Data Collection components, to other machines. The second issue relates to the DataServer. Because the DataServer is the only route that the Data Collection components can use to transmit the data to other components, the DataServer has to create child threads to handle the communication with every Data Collection component. If there are too many threads that are created, data dropout can occur as well. Several DataServer instances can be distributed across platforms or processors to solve this performance problem. This method needs the Data Collection component and all client components to be properly configured for each DataServer.

Distribution in the Data Layer also makes the system more reliable. Having separate Data Collection components for each medical device reduces the risk of complete system failure in case one or more medical devices fail.

An initialization strategy for the components of a distributed system is important too, because this may lead to a system deadlock. In the Data Layer, the components are not required to start in order (see more details in [48]). For example, the Data Collection components can start before and keep trying to establish connections until the DataServer is starting. This ensures that deadlock will not occur.

Task Layer

The current Task Layer contains only instances of the Alarm Detecting Client (ADC) which is designed to be a small and simple program. Users have a straightforward way to configure the ADC with parameters and events that need to be monitored, email of care providers who are interested in these events, and other properties as described in chapter three through one configuration text file.

However, because some important properties are needed such as dynamic configuration methods, multiple-parameter event monitoring, real-time notification, and so on, the current Task Layer as initially designed is limited. Moreover, although the ADC itself is small, modifying the current Task Layer is not simple since it is monolithic. The revised Task Layer was designed and implemented using Microsoft DCOM techniques which offer a simple process to develop distributed applications (it should be noted, however, that DCOM techniques have a steep learning curve). Designing a set of components with distribution in mind from the beginning enhances the expandability, capability, and performance of the system.

Three levels of DCOM servers and clients are grouped to form the revised Task Layer. The SSE Level is responsible for acquiring raw data from the DataServer and monitoring a single-parameter event. This prevents the DataServer from passing redundant data to the Task Layer when multiple ADCs request the same data items. The SSE Level is also designed to permit the easy addition of single-parameter events. The MSE Level monitors multiple-parameter events. When the SSE Level detects a single-parameter event, it notifies the MSE Level. Consequently, the MSE Level checks whether the single-parameter event, with or without other single-parameter events, triggers a multiple-parameter event. Users setup methods to allow the Task Layer to notify themselves in the Notification Level. In addition to email notification, the revised Task Layer provides two more notification methods, GUI notification through client, and beeper notification. Client development is open; developers are not restricted by any specification such as programming language or input methods except that clients need to be able to call DCOM objects in the Notification Level through DCOM interface, which is still under development.

About the initialization of the components in the revised Task Layer, DCOM also provides a simple initialization strategy. When the client is staring, DCOM will automatically start servers (i.e., for all three levels) as needed. The SSE Level that needs to communicate with the DataServer can also start before and keep trying to establish a connection like the other client components in the Data Layer.

Result and Current Status

The revised SIMON system was initially deployed in the VUMC trauma unit in September, 1998 on one bed. A second test bed was added in May, 1999. In April, 1999,

physicians and nurses started reviewing the data graphs on a SIMON-Trauma web site (see figure 8) during the course of patient care. They could view the graphs from a web browser running on bed side laptop computers or view them remotely from any authorized computer connecting to the network with appropriate security identification. As of October 29, 1999, data had been collected for 151 patients. Physicians and nurses began reviewing the data graphs at the bedside web browser in April 1999.

In August, 1999, the ADC was deployed and began detecting events and delivering daily patient status summaries to an ICU physician via email. A sample summary message sent to the physician is shown below.

Date: Tue, 28 Sept 1999
Subject: ALARM MAIL

Bed XXXXX Simon Alarms
from 09/27 5:00
to 09/28 5:00
Data collected for 22 hours 43 Minutes
ICP > 25.00 from 09/27 9:45 to 10:14
ICP > 25.00 from 09/27 10:36 to 10:52
ICP > 25.00 from 09/27 11:32 to 11:49
CPP < 60.00 from 09/27 22:04 to 22:41
02%Sat < 90.00 from 09/28 4:08 to 4:35
02%Sat < 90.00 from 09/28 4:36 to 5:00

The status summaries are a list of detected events ordered by start time. An email message includes start time, end time, bed, and total time period. The total time period is the total duration a heart rate signal was detected since patients are always instrumented for heart rate monitoring. Normally, the total monitored time is less than 24 hours, as patients leave the bed to undergo procedures. Also, data gaps of less than 30 seconds are ignored. If no event is detected, a short message indicating only the monitored time is sent to verify system operation. If there was no monitored data over the 24-hour period, no email is sent. Table 2 shows the events and information that were detected over the first two-month period. The events defined over a temporal interval, i.e., “CPP < 60 for 15 minutes”, describe events that are defined as a continuous interval greater than the time threshold. Gaps in the data that are longer than 30 seconds also occasionally divide the single event into separated events.

The process of expanding the system to a new bed and adding new devices was straightforward. After installing the new devices to the new bed, the other steps only require copying and setting the Data Collection components such as the HPDA, providing them with associated bed id, and start them. The process is dynamic because the DataServer and other running components in the system do not need to stop, and data from the new beds are acquired and monitored immediately. The more updated results and the most current status can be viewed in [17, 46].

Table 2: The table shows the events and information that were detected over the first two-month period. Number of patients shows the number of different patients that had the related parameter monitored for any duration of time [17].

Event	**Total monitored time (HH:MM:SS)**	**Number of events detected**	**Number of patients**	**Events per monitored hour**	**Events per patient**
SvO2 < 60 15 minutes	344:48:56	13	11	0.04	1.18
ICP > 25 15 minutes	768:19:29	81	13	0.11	6.23
CPP < 60 15 minutes	728:20:54	39	13	0.05	3.00

Future work and Discussion

Some existing components can be improved to enhance the system performance and extend its functionality.

Data Layer

Most of the components in this layer use CSocket to implement TCP/IP Socket communication. A lower level Winsock2 API can be used to improve performance and

reliability. UDP/IP protocol which is much faster but less reliable can be used instead TCP/IP protocol to improve performance. Therefore, more codes are needed to ensure reliability.

DataServer

As presented in chapter two, the DataServer constructs windows threads to handle a communication with its Data Collection components and data clients. Each windows thread communicates with other threads and its socket by using windows message. But a windows thread using its own message queue is substantially slower than a normal thread. An I/O completion port technique [43] introduced in Windows NT 3.5, together with a normal thread and variables synchronization, can be used instead of windows message queue to improve the DataServer performance.

The SSE Level

Adding a new type of a single-parameter event is a simple task. An event can be added to the SSE Level by constructing a new class inherited from the SSEObj. The new class needs only to implement algorithms to detect its type of event. The CSSEManager also needs to be modified to be able to construct an instance of the new type.

The Notification Level

Currently, the Notification Level is only partially implemented. It can receive event expressions, pass the expression to the MSE Level and notify a user client via its own user interface implemented for real-time testing. The COM interface still needs to be defined and the other notification methods need to be added.

Adding new components

While the system is running, new types of the Data Collection components and new type of data clients can be implemented and added to the Data Layer without difficulty. Also, the revised Task Layer using DCOM techniques makes it possible to develop custom client by simply following DCOM interface implemented by the Notification Level.

The revised SIMON system was designed to be an utterly dynamic and flexible system. There are limitless possible ways to improve the system capability. Finally, as newer computer technology both in hardware and software emerges, these new solutions that might improve the robustness, speed, and reliability of the system will have to be explored.

APPENDIX A

COMPLETE LIST OF SIMON PARAMETER

HP_CPP – HP Monitor [HP] Cerebral Perfusion Pressure

HP_CVP – Central Venous Pressure

HP_DBP – Diastolic Arterial Pressure (invasive from catheter)

HP_DPAP – Diastolic Pulmonary Arterial Pressure

HP_HR – Heart Rate

HP_ICP – Intracranial Pressure

HP_MAP – Mean Arterial Pressure (invasive)

HP_MPAP- Mean Pulmonary Arterial Pressure

HP_NBP_d – Non-invasive (cuff) Pressure - diastolic

HP_NBP_m – Non-invasive (cuff) Pressure - mean

HP_NBP_s – Non-invasive (cuff) Pressure – systolic

HP_RESP – Respiration rate determined from EKG electronic impedance measure

HP_SBP – Systolic Blood Pressure (invasive)

HP_SPAP – Systolic Pulmonary Arterial Pressure

HP_SpO2 – Pulse Oximetry

VL_AWRR – Ventilator Average Working Resp rate, via HP VueLink [VL] module

VL_BSA – Body Surface Area from Baxter

VL_CI – Baxter Cardiac Index

VL_CO – Baxter Cardiac Output

VL_EDV – Baxter End Diastolic Volume

VL_EDVI – Baxter End Diastolic Volume Index

VL_ESV – Baxter End Systolic Volume

VL_ESVI – Baxter End Systolic Volume Index

VL_FIO2 – Ventilator Fractional Inspired Oxygen

VL_O2EI – Baxter Oxygen Extraction Index

VL_PEEP – Ventilator Post End Expiratory Pressure

VL_PIP – Ventilator Post Inspiratory Pressure

VL_Pmean – Ventilator Mean Airway Pressure

VL_Ppeak – Ventilator Peak Pressure

VL_Pplat – Ventilator Plateau Pressure

VL_PULSE – Baxter Pulse Rate

VL_REF – Baxter Reference

VL_SI – Baxter Stroke Index

VL_SpO2 – Baxter Pulse Oximetry

VL_Sp-vO2 – Baxter Venous Oximetry

VL_SV – Baxter Stroke Volume

VL_SvO2 – Baxter Venous O2 Sat

VL_Tblood – Baxter Blood Temp

VL_TV – Ventilator Tidal Volume

VL_VQI – Baxter Ventilation Perfusion Index

Imed1_A_Conc – IV Pump 1, channel A Concentration

Imed1_A_Dos – IV Pump 1, channel A Dosage

Imed1_A_DrugName – IV Pump 1, channel A Drug Name

Imed1_A_RATE – IV Pump 1, channel A Rate

Imed1_A_VTBI – IV Pump 1, channel A Volume to be Infused

[Support multiple, 1-4 channel IV pumps]

CEN_MRN – Medical Record Number from Hospital Census System from

Oxim_SaO2 – Ohmeda Biox 3740 Pulse Oximeter (Oxim) Oxygen Saturation

USER_NOTE – Note from SIMON-Note

APPENDIX B

SIMON SYNTAX

This appendix shows syntax of SIMON name and expression used in the SIMON system. The SIMON name in the Data Layer is used as a parameter name sent and received from the DataServer. The Expression Syntax in the Task Layer is syntax of a multiple-parameter event passed to the MSE and Notification Level. Both syntaxes are presented in Backus-Naur Form (BNF) [47].

SIMON Name Syntax

param -> bedid_paramname

bedid -> simonstr^{+}

paramname -> *ivsource_ivdata*
| **HP_***hpdata*
| **VL_***vldata*
| **Oxim_***oximdata*
| **CEN_MRN**
| **USR_NOTE**

hpdata -> "based on HP monitor"
vldata -> "based on VueLink module"
oximdata -> "based on Oximeter model"

ivsource -> **Imed***ivnum_ivchannel*

ivnum -> **1** | **2** | **3** | **4**
ivchannel -> **A** | **B** | **C** | **D**

ivdata -> **Conc**

| Dos

| DrugName

| RATE

| VTBI

example of_paramname

HP_HR

HP_ICP

HP_MAP

VL_PEEP

VL_PIP

VL_Pmean

Imed1_B_RATE

Imed1_B_VTBI

Imed1_C_Conc

Oxim_SaO2

CEN_MRN

USR_NOTE

simonstr -> a..z, A..Z, 0..9

example of_param

10005T_HP_CPP

timestamp -> "the number of seconds elapsed since midnight (00:00:00), January 1, 1970"

archivefilename -> paramname.**d**
archivefileformat -> timestamp arvhivevalue

archivevalue -> **integer**
| float
| string

example of archivefileformat

HP_HR.d
954267761 93.00

Expression Syntax

expr -> *logicop*
| *param*

logicop -> **AND(***expr expr* **)**
| **OR(***expr expr* **)**

paramob -> **GT(***param, thresholdvalue, duration***)**
| **LT(***param, thresholdvalue, duration***)**

threasholdvalue -> **float**

duration -> "number in minutes"

example of expr

OR(AND(OR(GT(**10005_HP_SpO2**, **60**,
5)LT(**10005_HP_CPP**,**50**,**5**))LT(**10005_VL_SpO2**,**100**,**3**))
OR(GT(**10005_Oxim_PR**,**60**,**5**)LT(**10005_HP_MAP**,**50**,**5**)))

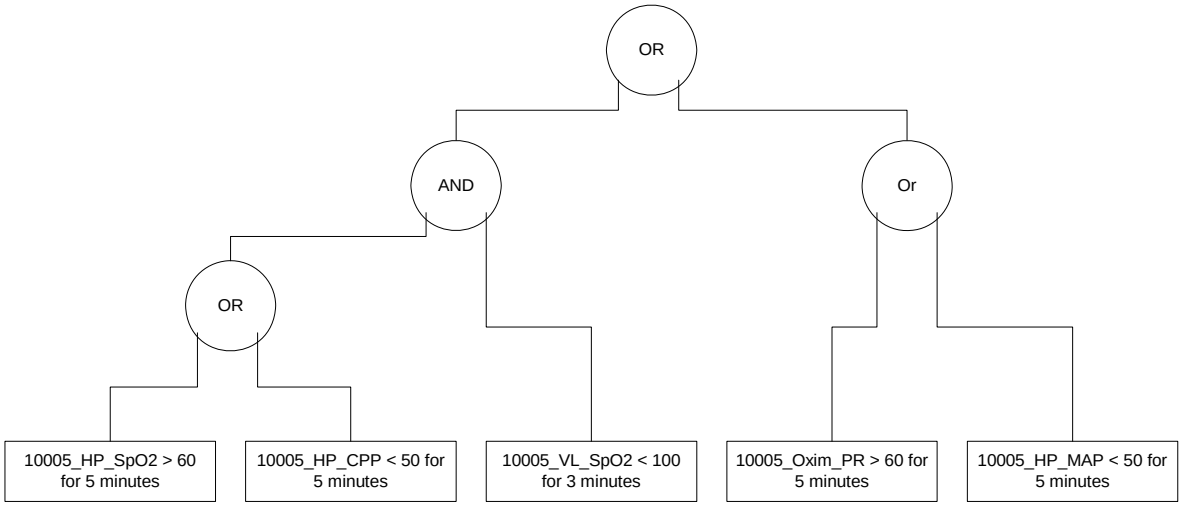

Figure 20: Tree of Expression

Class SimonNameSpace

SimonNameSpace encapsulates *param* string, and provides methods to retrieve any sub string. This class can be used in every component written in C++, especially in Data Layer components. STL string and STL vector are used in this class instead of MFC for a portability and performance purpose. The header file is shown below.

```
class SimonNameSpace
{
private:

      // typedef vector<string> StringList in "SimonDL.h"
      StringList m_SubStr;
      string m_sFullName;

public:
      SimonNameSpace(){};
      SimonNameSpace(LPCTSTR FullName);
```

```
        virtual ~SimonNameSpace();

        SimonNameSpace(const SimonNameSpace& NS);
        SimonNameSpace& operator=(const SimonNameSpace& NS);

        // Greater operator
        bool operator>(const SimonNameSpace&) const;
        // Less than operator
        bool operator<(const SimonNameSpace&) const;
        // equal to operator
        bool operator==(const SimonNameSpace&) const;

        operator const LPCTSTR() const;

        string GetFullName() const;
        string GetBedID();
        string GetString(int begin, int end);
        string GetDevice_Param();
        string GetDevice();
        string GetParamData();
};
```

REFERENCES

[1] LD. Hudson: "Monitoring of Critically ill patients," *Conference Summary on Monitoring ill patients*, 30:628-36, 1985

[2] R. Matthew Sailors, Thomas D. East: "Role of Computers in Monitoring," *Principles and Practice of Intensive Care Monitoring,* Part VII, Computers and Monitoring, Chapter 76, 1329-1354, McGraw-Hill Inc. 1998

[3] IEEE 1073 Medical Information Bus (MIB) Standards Committee Home Page http://grouper.ieee.org/groups/mib/

[4] Benoit M. Dawant, Patrick R. Norris.: "Knowledge-Based Systems for Intelligent Patient Monitoring and Management in Critical Care Environments," *The Biomedical Engineering HANDBOOK, second edition,* VOLUME II, Section XIX, 186, CRC Press, IEEE Press 2000

[5] I.J Hamowitz, P.P. Le, I.S. Kohane: "Clinical monitoring using regression-based trend templates," *Artificial Intelligence in Medicine,* 7(1995) 473-796

[6] Y. Shahar, S. Miksch, P. Johnson: "The Asgaard project: a task-specific framework for the application and critiquing of time-oriented clinical guidelines," *Artificial Intelligence in Medicine*, 14(1998) 29-51

[7] M. Dojat, F. Pachet, Z. Guessoum, D. Touchard, A. Harf, L. Brochard: "NeoGanesh: a working system for the automated control of assisted ventilation in ICUs," *Artificial Intelligence in Medicine*, 11(1997) 97-117

[8] C.P. Valcke, H.J. Chizek: "Closed-Loop drug infusion for control of heart-rate trajectory in pharmacological stress tests," *IEEE Transactions on Biomedical Engineering*, 44(1997) 185-196

[9] B.M. Dawant, S. Uckun, E.J. Manders, D.P. Lindstrom: "The SIMON project: mode-based signal analysis and interpretation in intelligent patient monitoring," *IEEE Engineering in Medicine and Biology* 12(1993) 82-91

[10] Baskar Raj D Ceri: "SIMONVIEWER – A WWW-BASED PATIENT MONITORING SYSTEM," *M.S. Thesis in Electrical Engineering, Vanderbilt University*, December, 1997

[11] B.M. Dawant, S, Uckun, E.J. Manders, and D.P. Lindstrom: "SIMON: A distributed computer architecture for intelligent patient monitoring," *Expert Systems with Applications*, 6(4):411-420, Oct-Dec 1993.

[12] E.J. Manders and B.M. Dawant: "Design of a dynamically reconfigurable critical care monitor," *Proceedings of the 19th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 1032-1035, Chicago, IL, 1997

[13] P.R. Norris, B.M. Dawant, and A. Geissbuhler: "Web-based data integration and annotation in the intensive care unit," *Proceedings of the 1997 American Medical Informatics Association Annual Fall Symposium*, pages 794-798, Philadelphia, PA, 1997. AMIA.

[14] W. Richard Stevens: *Advanced Programming in the Unix Environment*, Addison-Wesley Publishing Company 1992

[15] Eric J. Manders, MSEE, Benoit M. Dawant, Ph.D.: "Data Acquisition for an Intelligent Bedside Monitoring System," *Proceedings of the 18th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 957-958, Amsterdam, The Netherlands, 1996

[16] Patrick R. Norris, M.S., Benoit M. Dawant, Ph.D., and Karlkim Suwanmongkol: "Improving the SIMON Architecture for Critical Care Intelligent Monitoring," *Proceedings of the 1998 American Medical Informatics Association Annual Fall Symposium*, Philadelphia, PA, 1998. AMIA.

[17] Patrick R. Norris, Karlkim Suwanmongkol, John A. Morris, Jr., Antoine Geissbuhler, Daniel P. Lindstrom, and Benoit M. Dawant: *SIMON: A distributed architecture supporting real-time data delivery and event detection in intelligent critical care monitoring systems*, 2000

[18] Nuri Serdar Uckun: "AN ONTOLOGY FOR MODEL-BASED REASONING IN PHYSIOLOGIC DOMAINS," *Ph.D. Dissertation in Biomedical Engineering, Vanderbilt University*, August, 1992

[19] Introduction to TCP/IP
http://pclt.cis.yale.edu/pclt/COMM/TCPIP.HTM

[20] Herbert Schuldt, Frank Crockett: *MFC Programming from the ground up*, Second Edition, Osborne/ Mcgraw-Hill 1998

[21] Microsoft Foundation Class Library and Template
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcmfc98/html/vcmfchm.asp

[22] Herbert Schildt: *STL Programming from the ground up*, Osborne/ McGraw-Hill 1999

[23] Charles Petzold: *Programming Windows*, Fifth Edition, Microsoft Press 1999

[24] The source for Java™ Techonology
http://www.java.sun.com

[25] The Source for Perl
http://www.perl.com

[26] JavaScript Developer Central
http://developer.netscape.com/tech/javascript/index.html

[27] Structured Query Language – a whatis definition
http://whatis.techtarget.com/definition/0,289893,sid9_gci214230,00.html

[28] *HP Component Monitoring System, RS232 Computer Interface Programming Guide (Option #J13)*, First Edition, Hewlett-Packard 1992

[29] Etherlite
http://www.digi.com/solutions/termsrv/etherlite.shtml

[30] *HP M1032A VueLink External Device User's Booklet*
5th Edition, Hewlett-Packard 1996

[31] *HP 1032A VueLink Module Handbook*
Edition 3, Hewlett-Packard 1995

[32] VueLink Interface
http://www.healthcare.agilent.com/patient_monitoring/cgi-bin/show_product.pl?VueLink%20Interface

[33] *Servo Ventilator 300/300A, Firmware version 2.x, Computer Interface Reference Manual*, 2nd English Edition, Siemens-Elema AB 1997

[34] *Functional Specification, C2 Communications Protocol*
Revision 5.2, IMED Corporation 1991

[35] *Oheda Biox 3740, Computer Interface*, Datex-Ohmeda 1988

[36] Java[TM] Foundation Classes
http://www.java.sun.com/products/jfc/index.html?frontpage-javaplatform

[37] Client-Side JavaScript Guide: LiveConnect Overview
http://developer.netscape.com/docs/manuals/js/client/jsguide/lc.htm

[38] Microsoft COM Technologies
http://www.microsoft.com/com/

[39] Thuan L. Thai: *Learning DCOM*, O'Reilly & Associates, Inc. 1999

[40] Chris Corry, Vincent Mayfield, John Cadman, and Randy Morin: *COM/DCOM Primer Plus*, Sams Publishing 1998

[41] Richard Grimes: *ATL COM Programming*, Wrox Press 1998

[42] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides: *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison Wesley Longman, Inc. 1995

[43] Jeffrey Richter: *Advanced Windows*, Third Edition, Microsoft Press 1997

[44] The Winsock Channel at Stardust.com
http://www.stardust.com/winsock/

[45] James O. Coplien: *Advanced C++: Programming styles and idioms*, AT&T Bell Telephone Laboratories, Addison-Wesley Publishing Company 1992

[46] SIMON Project home
http://simon.project.vanderbilt.edu/

[47] Michael L. Scott: *Programming Language Pragmatics*, Morgan Kaufmann Publishers 2000

[48] Karlkim Suwanmongkol: *SIMON system technical paper*, 2001

[49] Process Trellis
http://www.cs.yale.edu/Linda/process-trellis.html

[50] The MAITA Project
http://www.medg.lcs.mit.edu/projects/maita/