A Software Architecture to Support a Large-Scale, Multi-Tier Clinical Information System

Joel A. Yungton, Dean F. Sittig, Paul Reilly, John Pappas, Steve Flammini, Henry C. Chueh, Jonathan M. Teich

Clinical Systems Research & Development
Partners Healthcare System, Boston, MA

ABSTRACT

A robust software architecture is necessary to support a large-scale multi-tier clinical information system. This paper describes our mechanism for enterprise distribution of applications and support files, the consolidation of data-access functions and system utilities stored on the data access tier, and an application framework which implements a coherent clinical computing environment. The software architecture and systems described in this paper have been robust through pilot testing of our applications at Massachusetts General Hospital.

INTRODUCTION

Partners HealthCare System (PHCS) is an integrated healthcare system founded in 1994 by Brigham and Women’s Hospital (BWH) and the Massachusetts General Hospital (MGH). Both hospitals brought large, complex computer systems to the table and each offered vastly different solutions to similar problems. In any merger, the decisions regarding which systems should be combined, which should be replaced, and which should be removed altogether must be made. The decision to combine two large hospital information systems rather than replace or throw one out leverages the strengths of each system while maintaining the respective investments. Once the decisions have been made, the work of building an architecture that supports the integration of disparate systems at the client begins.

In recent years, technology has moved closer to distributed networked systems and further away from centralized mini and mainframe systems. For large-scale integration, the strengths of networked distributed systems are key: flexibility, scalability, agility, and support for bridging disparate systems via custom interface layers. Yet, despite well-developed networking technology, the implementation of an enterprise-wide software architecture is far from trivial. In the early analysis stage, we identified the following issues and goals. First, with nearly 20,000 workstations in the network, software distribution alone could require a small army. With most application development efforts taking place in-house, timely and accurate software updates were deemed critical. The new architecture should streamline the delivery and maintenance of software systems for 20,000+ clients. Second, a solid data access tier comprised of software services and data access routines would have to be designed and built. Designs at this level should focus on reusable software modules that would minimize duplication of developer effort and maximize cross-application interoperability. Third, a diverse end-user population demands an intuitive, consistent clinical computing environment1. In the absence of a unifying force, new applications would otherwise take on their own look and feel, leaving the end-users to sort out a myriad of different styles and functionalities.

This paper describes three key issues that Partners faced in building a software architecture to support a large-scale multi-tier clinical information system. The first deals with enterprise distribution of applications and support files. The second describes the consolidation of data-access functions and system utilities stored on the data access tier. The third describes an application framework that implements a coherent clinical workstation.

SYSTEM DESIGN

Software Distribution

A network architecture uses a centralized program distribution model where applications are maintained and executed from a central network
Software distribution is trivial: the client downloads and runs whatever is available on the server. This occurs each time an application is started, as the client does not store executables locally. Conversely, a client-server architecture relies on local executables. This is an important feature of client-server as it frees the server from the menial task of program downloading and execution. It does however introduce the problem of program distribution—that is, the process of how executables get to each and every workstation in a network. Furthermore, maintenance updates and patches are actually much more complicated because knowing exactly which version of a particular application and associated dependent files resides on a client at any given time is a challenge. The problem of initial distribution is fairly obvious and can be handled with off-the-shelf software such as Microsoft Systems Management Server, which relies on a “push” technology to distribute files. With push technology, files on the server are sent to the client when the server wants to send them. A drawback of this scheme is that it is typically initiated during a client reboot or calendar-scheduled event. This does not bode well for hospital-based workstations that may be in need of a critical update, yet have not been rebooted for days.

To solve this challenge, we have developed a hybrid software distribution model that combines desirable features from both the network and client-server models. Namely, a central server handles software distribution, but software updates are delivered just in time on an as-needed basis. The client maintains a local program cache that is always checked first when a user starts an application. If the application is current, it runs immediately. If the application is out of date or does not exist on the client, the server will send down an update as necessary. The result is a distribution model that harnesses the power of a centralized server, yet exploits the speed of a local cache.

The PHCS Version Control/Version Console and PHCS Launcher provide the mechanism by which all Partners workstations stay up-to-date with the latest application program revisions. The PHCS Version Console resides on a network server and provides a front-end to the Version Control database. The remainder of Version Control consists of versioning software and two dedicated network disk areas which are used to hold distributable programs and their supporting files. The PHCS Launcher is part of the Partners common desktop and as such is installed either manually as part of the new workstation build process, or via SMS. The PHCS Launcher thus resides on every workstation. It communicates with the Version Control database each time an application is executed, sending information such as the workstation type and software version of the application being started. The launcher is said to use “pull” technology because a request for incremental updates is generated by the client and occurs each time an application is started. Since the client drives the requests for updates, it is almost always in sync with the latest releases.

Two key features of this system are the ability to define projects and their dependencies, and the ability to specify the distribution target. A project typically consists of an application program and any directly dependent files. Additionally, resources that may apply to multiple projects are separated out into shared libraries but may also be included as a dependency. A project, together with its dependencies is bundled as a “release” for a specific target workstation type (Alpha, Beta, or Production) and assigned a version number.

For example, a Visual Basic application that uses Partners M/Services (a collection of software utilities that provides access to MUMPS legacy data) can specify M/Services as a project dependency. Suppose further that this project is bundled into an Alpha release. When an end user starts (“launches”) the application from a client, the version console first verifies the workstation type then ensures that the latest version of the application available for the type is on the workstation and that the correct version of dependent files (M/Services) are also on the machine. If any files included in the Alpha release are out of date, then Version Control will send down the correct software and perform any application-specific installation tasks necessary. By specifying the release target, software intended, for example, for QC testing can be automatically sent to workstations designated as Beta, whereas workstations designated as Production will not receive the new release. The project definition structure combined with the ability to specify a distribution target gives tremendous leverage to those concerned with initial software distribution and maintenance releases.

**Data Access Tier Objects and Services**

Partners, like many companies tackling the large-scale integration of systems—both old and new—has
been faced with the challenge of enterprise-wide data consistency and data access. There was no existing common denominator that would bring together data from around the enterprise, nor was there any common format for the display and handling of data once it was retrieved. Furthermore, it seemed inevitable that additional systems would need to be integrated as Partners grew. Therefore, the corporate strategy became one of adding an abstract “data access” tier which would provide common data objects and services to client applications while hiding the details of disparate back-end systems.

Partners chose the Microsoft Component Object Model (COM)\(^3\) to implement the desired architecture. COM is robust, easy to use, relatively fast, and has allowed application development to continue in parallel with middle tier development. Conceptually, the data access tier could reside anywhere in the network, but we have chosen to distribute data access servers to the client machines for better performance and to simplify the model. In the simplest terms, this architecture allows Visual Basic applications to get data from virtually any system in the Partners network without having to know where the data comes from. Of course, there are limitations. For example, client applications can only make requests for types of data that the data access tier knows how to retrieve (i.e. Partners is not yet using a universal query language that would return data based on an arbitrary request). However, with careful analysis and design, we have identified key objects and services that are having a noticeable impact on developer productivity and system agility.

The Partners-wide objects that are having the greatest impact are the PatientObject, UserObject, OrderEntry-based objects such as Order, Test and Medication, Service-based objects such as Observation, Procedure, and Therapy, and Results-based objects. The most widely used services include UserSecurity, which performs various UserObject-related functions including auditing and application authorization; PatientLookup, which returns a PatientObject; M/Services, which provides access to MUMPS-based data; and PCIClientManager which provides access to MGH data stored under Tandem Nonstop SQL. Each object and service has been analyzed and built in response to specific needs, yet each has been designed to provide maximum functionality and flexibility for the enterprise. In many cases, an inheritance relationship has been designed in so that fields like DateOfService are available at all levels of a class hierarchy\(^4\).

The true power of the objects and services is realized on two fronts: client-to-data access tier communications, and client-to-client communications. The first is most obvious. Services that reside on the data access tier have a well-defined interface and return data in the form of well-known objects. In this case, “well-defined” and “well-known” mean that the format of callable routines, their parameters, and data access methods have been agreed upon and are virtually “set in stone”. Modifications that impact multiple systems require justification and approval. In addition, most object and service documentation is online and is easily accessible by developers. Services like PatientLookup, and UserSecurity can simply be plugged into an application as needed. A reliable services interface allows developers to concentrate on their specific application rather than having to continually re-invent the data access routine, or the user validation routine.

By using an abstract data access tier, system agility is greatly enhanced because front-end processing is completely detached from actual back-end processing. A front-end GUI or back-end server could be swapped out without necessarily affecting the other. This has been demonstrated by several projects including one completed by MGH's Laboratory of Computer Science. The project entailed building a web-based clinical information system viewer. The web server interfaced with Partners data access servers to perform user authentication and authorization as well as to retrieve clinical data\(^5\). Other Partners efforts which have made use of data access tier objects and services via intranet include a web-based enterprise phone directory, and a Longitudinal Medical Record application that pulls together data from multiple sources. Another example of agility was demonstrated during a recent software pilot when a back-end process was re-directed to look first to a cached disk area before attempting a record retrieval. The only noticeable change for the end users was a five-second improvement in search times. Client software did not have to be modified.

The second area that has shown an increase in productivity and reusability is client-to-client communications. This has been demonstrated primarily through the use of data objects. For example, the PatientObject returned from the PatientLookup service may be passed on from one application to another. The task of transferring fifty data fields from one process to another is simplified.
by passing the entire PatientObject. Another area where this type of reuse has been demonstrated is with the UserObject returned from the UserSecurity service. A high-level application may validate a user, then pass the UserObject on to sub-applications thereby simplifying security access within that hierarchy. The concept of passing data structures around is not new; however, with COM a new level of openness has been achieved which allows easy inter-process communications and provides a path to connect applications written in different languages such as Visual Basic and C++.

As easy to use as COM is, there is one drawback that had to be addressed: data access server security. Recall that the data access tier though logically separated from the client is physically located on the same machine. Any savvy user who knows a little about ActiveX Servers knows that they can see the interface, connect to a local server and use its services through programs like MS Access and Excel. In fact, reuse in this manner is encouraged, but it must not occur without a security model. There are simply too many confidentiality holes that are opened otherwise. For example, the savvy user may create an instance of the PatientLookup and start retrieving PatientObjects. PatientObjects could then be used to retrieve patient results and security has just been breached.

The solution was for us to implement a security scheme that prevents unauthorized use of data access servers. At a high level, applications need to be authorized, which basically means that they are registered in a database of legitimate systems. All applications that will be used throughout the enterprise must be authorized in this manner. These applications must also be installed in the version database and launched, rather than executed locally. Launched applications receive a token called the Application Launch Key (ALK) that they must pass back to a security server to verify that they have in fact been launched. If the launch sequence succeeds, a second security token called the Server Launch Key (SLK) (which the data access servers need) is generated. Upon initialization, a data access server will make a call into the security server and ask for the current SLK. If the SLK returned by the security server matches the SLK made available by the application, the system is deemed secure. Without a validated SLK a data access server will not function.

**Application Framework**

The Clinical Application Suite (CAS) is an application framework that software developers use to house their applications. CAS performs three main tasks: 1) Merges multiple clinical applications into a single visual and functional context for the user; 2) Maintains a single CurrentPatient object across all applications; 3) Consolidates common system services.

As an application framework, CAS takes GUI design one step beyond mere screen layout: it actually handles a certain amount of application functionality. Externally, CAS provides a visual front end for all clinical applications that we have chosen to integrate. When CAS is running on a workstation, the user sees a display panel across the top of the screen, and a button panel down the left side. These two panels, which comprise the visual element of CAS, are anchored on the screen and are always visible while the remaining screen area is available for a clinical application’s use and will change depending on which application is currently visible. Each of the buttons will switch the user to a different application. CAS handles the task of allowing the user to switch back and forth among multiple running applications as well as starting (or shutting down) an application. Because of its persistence on the screen, CAS provides a constant point of reference for the user. The result is a large, highly modular system of clinically related applications that feels like a single application.

Internally, CAS provides an application programming interface (API) that specifies how information will be shared between CAS and its applications. An application is said to be CAS-compliant if it adheres to the CAS’s API. Chief among the services that CAS provides is a series of messages which coordinates the CurrentPatient object across all CAS-compliant applications. For example, applications may request the CurrentPatient object and may, via protocol, change the CurrentPatient. CAS processes these requests and notifies all CAS applications of the change. This type of cross-application patient coordination is becoming more prevalent industry-wide.

CAS also consolidates common system services and information whenever practical. For example, many clinical applications need to perform a task that relates to a single patient. Outside of the CAS, the application would need access to the Partners
PatientLookup services, and would probably want to display some of the current patient's information as well. Rather than have every application that needs a PatientObject connect to the PatientLookup directly, the CAS provides a single connection to PatientLookup that each CAS-compliant application may utilize. CAS also displays information on the current patient in its top display panel. This not only frees an application from having to maintain a connection to a system service, but also makes efficient use of valuable screen space by freeing developers from having to display the patient information themselves. Additionally, the user always knows where to find the current patient—the information does not move around from application to application. Thus, both developers and users benefit from such consolidation.

**PRELIMINARY RESULTS**

The architecture described in this paper has been completed and is being pilot-tested by over 100 clinicians at MGH. Our tests have shown that launching an application for the first time on a workstation takes between 20 and 90 seconds, depending on the size of the application and the number of dependent files. Subsequent updates typically take between 20 and 35 seconds. A "no update needed" determination can be made in less than one second. Launch benchmarks were performed on an Intel-based P90 workstation with 32 MB RAM.

The data access tier has proven quite reliable. Its true agility will soon be tested with the implementation of our new Clinical Data Repository. Once this enterprise-wide database is in place, the data access tier will be pointed away from the Tandem Non-Stop SQL system to the new MUMPS database.

The CAS concept has proved to be very reliable and easy for clinicians to use. During a 15-day pilot in an outpatient clinic, 24 clinicians logged into the CAS a total of 368 times and accessed at least one of the three currently available clinical applications. The average user looked at 3 patients per session and utilized 2 applications. Users accessed the online help function a total of 3 times during the pilot.

**CONCLUSION**

Combining two large hospital clinical information systems is a challenge, but it can be accomplished using distributed networking technology when combined with a strategic software architecture. PHCS has realized that a successful implementation requires much more than just physical hardware and wiring—it requires a software distribution plan, a solid data access tier that emphasizes modularity and reuse, and a software framework that promotes application integration and resource consolidation.

The software architecture and systems described in this paper have been robust through pilot testing of our applications at Massachusetts General Hospital. We will continue to watch and evaluate them during further rollouts in 1998 and beyond.

**References**


http://www.microsoft.com/smsmgmt/default.asp (7/10/98)

http://www.microsoft.com/com (7/10/98)


