

Data-Centric Medical Information Systems

Copyright © 2002 Craig F. Feied, MD
and Mark S. Smith, MD

We present a model for emergency department computer systems that will support clinical practice, education, research, and administration via a data-centric model that is distinctly superior to the traditional models used in older commercial systems. We call this the distributed data-centric model for medical computing.

The monolithic model

Most medical computer systems available today were designed and built according to a 'monolithic' model that arose more than 25 years ago and reflected the tools and resources of that era. In this monolithic model, a single large computer program handles all aspects of the overall process (see table 1). This monolithic program ran on a monolithic hardware system that included not only the computer itself, but also every external expression of the system: the input ports, wires, terminals, printers, on-line and off-line storage devices, modem connections, and everything else that was used in the overall process.

Table 1.
Functions integrated into a single monolithic computer system

- Accepting and validating incoming data from other computer sources.
- Constructing & displaying data entry and edit screens.
- Validating data that has been entered into a field on an entry or edit screen.
- Sorting the data.
- Storing the data.
- Backing up the data.
- Removing old data to an off-line archive.
- Managing all aspects of archived data.
- Constructing & maintaining indexes for the data.
- Designing & running reports using the stored data.
- Executing periodic reports and other periodic functions.
- Formatting reports for a printer and sending the data to the printer.
- Constructing and displaying data query screens.
- Creating and maintaining a list of valid users.
- Providing security for user access.
- Sending data to another computer system.
- Every other function related to the overall process.

When the monolithic model arose, there were few (if any) industry-wide standards for hardware, operating systems, or interfaces. Every hardware system was proprietary. A disk drive from one manufacturer wouldn't work with a drive controller from another. Each manufacturer invented unique interfaces to control video monitors, printers, memory cards, and other peripherals. There was no common bus to allow plug-in cards to be used interchangeably from computer to computer. A floppy disk formatted on one machine couldn't be read on another machine from a different manufacturer. Data was stored in proprietary formats that were guarded as trade secrets, and there was no standard for data interchange. Every application development project required custom programming on proprietary hardware, running a proprietary operating system, and using proprietary versions of a programming language. Printers and other devices were driven using proprietary protocols, and all interfaces with other systems were developed using unique hardware and software solutions that were utterly nonstandard. Although these systems represented the state of the art, investment in such a system inevitably meant an expensive multi-year commitment to ongoing development, debugging, and maintenance of a system that could only be cost-justified based on a fifteen to twenty year life cycle.

Buy versus build

In the beginning, there were no commercially available applications to support medical practice, medical billing, laboratory systems, or any of the other functions needed in a hospital or a medical practice environment. A hospital wishing to computerize any of its functions had no choice but to develop custom in-house software solutions using a large proprietary computer system. Unfortunately, those who tried in-house development found a rocky path. All of the costs of initial development, ongoing maintenance, and upgrades had to be borne by the developing institution, instead of being spread out among a large base of installed customer sites. Project design was guided by an a priori perception of the needs of medical personnel that was not based on any previous experience with medical computing. Development tools were slow and cumbersome, and many system development efforts never survived to implementation. In-house systems that survived to be implemented often were abandoned shortly after installation because they failed to meet the real work needs of end-users. When the first commercial medical software solutions became available for these behemoths, it was easy to see that buying a commercially developed and supported system was faster, easier, cheaper, and more predictable than building a custom system. The prevailing dogma became 'buy, don't build'.

Benefits of purchasing an established system

Many things have changed since the early days of medical computing, but there remain a number of benefits to purchasing an established system. Many worker-years of design and development have already been invested into such a system, and the cost is amortized among all of the purchasers. The costs of ongoing development, bug fixes, and general software maintenance are likewise integrated into the price, and are thus shared among all of the purchasers of the system. By the time a product is offered for general sale, many improvements and enhancements have been applied to the original design. There is an installed user base to demonstrate that the product really exists and really does do what it is supposed to do. The delays, struggles, compromises, and cost over-runs inevitably associated with new product development are already over and done with.

Risks of purchasing an established system

Unfortunately, there are also many disadvantages to purchasing an integrated computer solution. Certain problems are nearly universally present when a complicated business is forced to depend on a commercially sold and supported monolithic computer system.

Poor business model fit

One common problem stems from the fact that commercial software is primarily designed to meet the sales and marketing needs of the vendor. Meeting the specific needs of any individual customer is important to the vendor only to the point where the customer is willing to pay the purchase price. Once the decision to buy has been made, the system has met the vendor's needs. From that point forward the process might be described as WTSIWYG computing (what-they-sell-is-what-you-get). A purchaser usually is constrained to accept the software and hardware package more or less 'as is', and the promise of extensive customization after purchase is rarely fulfilled. Features and functions will have been optimized to meet the needs of the marketplace, rather than being optimized for the specific needs of an individual purchaser. In most cases, the buyer is forced to change business and practice models to fit the software, rather than having a software solution tailored to fit the existing business model.

Slowness to change

Ownership of the source code remains with the vendor, thus the end-user is unable to modify and update the software when new changes are introduced into the business or practice model. Requests to the vendor for customization will be expensive, and will compete for attention with the needs of other customers and the needs of the vendor to release new products. The turnaround time can become very long even for small or simple changes. If the software cannot be quickly and easily changed to accommodate changed business practices, users may be forced to continue obsolete and counterproductive practices, or else may pay the high exit costs associated with abandoning an installed system on which the business depends.

Obsolescence

Hardware platforms today undergo a significant change every 6 months. Operating systems change every year. Those who purchased monolithic systems that were state-of-the-art 6 years ago now have a large investment in green-screen, character-based dinosaurs with little capacity for change to meet today's needs.

Life cycle of proprietary solutions

Medical business and practice models today are undergoing very rapid change, yet many commercially sold and supported medical packages still run on proprietary hardware using proprietary protocols and connecting through proprietary device interfaces that have changed little in the past fifteen years. In a typical monolithic application, modifications and patches change the system over the years in unplanned ways, until finally it becomes difficult or impossible to modify the system in one area without introducing problems in another area. The features and functions of monolithic applications are spread throughout an extremely large and complex program. It is impossible to cut the program apart into component parts, so the entire program live or dies as a monolithic whole. If the operating system or the computer hardware becomes obsolete and is no longer supported, it may be difficult or impossible to migrate the program to a new operating system or even to a new computer running the same operating system.

No monolithic computer hardware and software system being sold today can reasonably be expected to meet the needs of users fifteen to twenty years from now, yet many will not achieve any reasonable return on investment in a shorter time frame. Even when the underlying architecture of a system is based on more modern hardware using open standards, a monolithic software solution still can create future problems if it

carries a purchase price and cost of adoption too high for its projected life cycle, if upgrades and add-ons are expensive or difficult to come by, or if the costs of abandonment of part or all of the system are high enough to keep users 'locked in' when the system no longer meets their needs.

Challenges to the monolithic model

Many new developments have changed the face of computing since the monolithic model came to dominate the landscape. A large number of widely-accepted standards have emerged in hardware, operating systems, interfaces, input/output devices, data storage systems, data query and retrieval languages, and software interactivity. Modularity has been widely adopted at every level of systems development. Because of these new developments, open standards, and modularity, it is now possible to design and implement a system that is optimized to take advantage of recent progress that has already occurred and still offers a clean mechanism for continuing to take advantage of each new improvement that arrives in the future.

Development tools

Newer software development tools are orders of magnitude more powerful than those in use a few years ago. A typical database project that would have required two years of intensive development effort in the 1980s now can be completed with fewer resources in six months or less. Some of this is due to the introduction of object-oriented programming methods, some of it is due to improved integrated development environments (IDEs) and some of it is due to the existence of open standards that allow the use of externally-developed components to speed the development of a custom project.

System costs

Not only is the cost (in time and dollars) of initial development of a medical software system a fraction of what it once was, but the cost of the machines and networks on which such a system will run is also a fraction of the cost of yesterday's mainframes. After the system is up and running, the ongoing cost of modifications and maintenance will be much less than in the past. Even the cost of end-user training is lower than it used to be, partly because modern software is much easier to use and partly because modern users are much more familiar with computer systems.

Open standards

Open standards are standards for which the specifications have been openly published and released for use by any manufacturer without the need to pay license fees. Successful open standards have been adopted by multiple hardware manufacturers, with the result that many different software and hardware solutions are compatible with the standard and thus with each other. Market forces ensure that systems based on open standards are ultimately less expensive and more likely to be up-to-date than proprietary solutions.

Modular system design

Modularity is important to overall system cost and performance at every level, from the integrated circuits that populate computer circuit boards to the accessory boards that plug into the computer and the monitors and printers that provide output. In modern software design, many of the component objects used to create a program can now be purchased as pre-written modules to be mixed and matched. Modular programming leads to increased development speed and robustness because well-engineered modules can be re-used in more than one project. Most important for system longevity is the opposite attribute: the ability to throw away a module in favor of a newer one that performs the same functions better, faster, more cheaply, on a different machine, or with an improved feature set.

At the system integration level, modular design frees us from the constraints of a monolithic application that lives or dies as a monolithic whole. A modular system can be more easily migrated from operating platform to platform and from machine to machine. Modular systems can be upgraded a piece at a time. New features can be added without destabilizing the old. An entirely new front end can be pointed at the same database, or an entirely new database can be adopted to serve data to the original front end.

Modularity also facilitates the development of robust systems by supporting redundancy. For example, it is possible to establish multiple sources and pathways by which to receive the same data. Secondary or tertiary copies of the data can simply be discarded if the primary copy has arrived successfully. A similar redundant approach, utilizing automatic failover to a backup hardware or software module, may be applied to every part of the system.

The data-centric model

The possibility for rapid and inexpensive development, low hardware and networking costs, system-wide modularity, and the existence of open standards have produced a radically new environment for medical computing. The resulting improvements in computer hardware, operating systems, and software have changed the equation so much that it is now reasonable to consider either building a custom system, assembling a system from commercially available components based on open standards, or a combination of the two – buying some parts of a system and building others. It is now possible to buy a base data storage system, to mix and match components from different vendors, to order custom modules, attributes, and functions where needed, to replace components as they grow obsolete, and to program new input and output modules as needed. It is this flexibility that makes the distributed data-centric model of medical computing possible.

Under a model of distributed data-centric medical computing, a small departmental data server (or a cluster of small servers) is established using off-the-shelf hardware and software. Data is acquired through a series of standard interfaces to existing systems and to end-users. The data is parsed and modified by independent processes, is stored in an industry-standard database module, and is delivered from the database in response to queries written using a standard query language. Data is formatted as desired for display through native client browsers, web-based browsers, report generators, and other output systems. Events can be triggered whenever certain conditions are met. Filters and sorts can be applied to the data to select any desired subset, which is then delivered by standard protocols to the requesting module. Any component of the system can be replaced at any time if a better or faster alternative becomes available (see figure 1).

The system is considered data-centric because each atom of data lives independently from the moment it enters the system until it is discarded or finds a resting place in the database. Incoming data flows through a series of data-atom-oriented processes, each one separate and distinct from the next, until it arrives in some repository from which it is available to other independent processes by means of some query mechanism. The incoming processes, the storage data column structure, and the client processes that use the stored information all are defined around the data itself.

Modular processes in the data-centric model

Data is handled by a large number of modular processes interacting according to recognized standards. Each process is independent of the others, and each one can even be running on a different machine (see table 2). Each of these processes is performed by an independent module, and it is possible for many such operations to occur simultaneously. At any moment, new data is arriving from multiple different sources by different routes, is being processed according to a variety of data-specific rules, and is being stored in multiple locations according to rules governing the structure of the database. At the same time, multiple front-end processes are querying the database, recovering data elements, combining and reformatting them, triggering alarms, displaying formatted data, printing reports, and passing data along to other systems.

Table 2.
Independent modular processes in a data-centric model.

- Data from some source arrives via a standard protocol.
- Data is accepted as valid or rejected as invalid according to specified rules.
- Data is modified as needed by extraction, processing, or combination with other data.
- Data is stored in a database field or is discarded based upon another set of specified rules.
- Data is retrieved from storage by any authorized process by means of a standard query.
- Data is formatted, processed, or combined with other data.
- Data is displayed, printed, shared, or used to trigger some other event or process.

Table 3.
Current standards used for interconnection, communication, and storage in this example of a standards-based data-centric model.

- TCP/IP (Transmission Control Protocol / Internet Protocol)
- Sockets
- FTP (File Transfer Protocol)
- Ethernet
- HL-7 (Health Level Seven standard for medical data transfers)
- XML (Extensible Markup Language)
- SQL (Structured Query Language)
- ODBC (Open DataBase Connectivity)

Example

A real example taken from a working system (Active Query) that has been in operation for three years in the emergency department at the Washington Hospital Center (Washington, DC) and that is also in operation at Franklin Square Hospital (Baltimore, MD) serves to illustrate. In this system, data from a variety of legacy "smokestack" systems is delivered to the emergency department's departmental data repositories over a standard network and is processed, stored, and retrieved through a series of standards-based interconnections between modular processes running on a variety of different machines.

The underlying network architecture conforms to industry standards: all interconnections are made using the TCP/IP network protocol, industry-standard Ethernet hardware, and standard category 5 cable. In a transaction mode, incoming data messages are received by an interface engine module using a sockets connection (part of the TCP/IP industry standard). In most cases, incoming messages conform to some degree to an HL7 data format. In a batch mode, incoming files are received by a standard File Transfer Protocol (FTP) module and are then picked up by an interface engine for further processing. Whenever possible, batch-mode data is formatted or preformatted using the XML standard for data interchange.

For each message that arrives, the interface engine generates an independent 'thread' (a part of a computer program that can execute asynchronously with other parts of the program) to handle the message. At any moment, an interface engine may have multiple active threads processing multiple messages that could originate from many different sources. When a message arrives and is given to a new thread, the thread checks to see that the incoming message is a valid one, that it was sent from a valid and authenticated source, and that it contains a type of information that the sending machine is authorized to send and that the engine is authorized to receive – all based upon a list of rules that can be configured by the system installer. If the message passes all validity tests, the interface engine thread accepts the message and sends an acknowledgment receipt to the sending machine (often another interface engine or a hospital mainframe interface). The thread then classifies the message and stores it in a local "parsing queue" in an SQL-compliant database, where it will be accessible to the next group of modules in the Active Query system. When a thread is done with its work, it kills itself, but before terminating, it also updates a 'message stream status' database containing information about the number

of messages of each type that have been received, the number that have been rejected, the current status of the interface for each message type, and other system console data. One or more console monitor modules (running on different machines) read the message stream status database and show the current and recent status of each message stream.

At Washington Hospital Center, many different interface engines run simultaneously, each one receiving and acknowledging data from a different source. Some of the interface engines are running simultaneously on a single shared computer, but others must handle so much data that they require dedicated computers in order to keep up with the volume of incoming messages. If necessary, a single data message stream can easily be split into two or more sub-streams, and each sub-stream can be given to a separate engine for processing. A separate interface engine module is instantiated to handle each of the following data streams: demographic information, lab and x-ray orders, lab results, x-ray text results, x-ray images, CT and MRI images, medications given, patient location (from a radio-frequency tracking system), vital signs (from a network of cardiac monitors), billing and collections information, historical patient data, and autopsy reports.

Once messages have been received, authenticated, acknowledged, classified, and stored for further processing, an entirely new tier of "message parsing modules" takes over. Each parsing module has complete responsibility for creating and modifying the contents of a single field or a group of conceptually related fields in the core database. The rules that govern the processing of messages and the ultimate contents of the database table columns are written in a simple and widely distributed scripting language that can be replaced or upgraded without affecting anything except the scripts themselves. These rules are easily modified, and new ones may be scripted by the database administrator as needed whenever a new data source is added to the system. Each parsing module is independent of every other. Parsing modules manage their own connections to the parsing queues from which they get their raw messages and to the core database in which they store the field-specific data they generate. Each parsing module uses SQL commands to request messages as needed from any queue to which it subscribes. SQL commands are also used to mark messages done when they have been processed, and to put the processed data into the correct field in the core database.

A special class of parsing modules examine any desired message streams and apply scripting rules to fire

system events, or "alarms," that can trigger any internal or external action or process that may be needed. Critical results may be emailed, faxed, or telephoned to any desired recipient. Armbands and face sheets may be automatically printed. Orders may be automatically generated. These scripts and the events they trigger can be arbitrarily complicated. For example, a message containing an order for a teratogenic medication might be processed by a script that first looks to see whether the patient is female, of child-bearing age, has no history of sterilization, and has not yet had a negative pregnancy test. If those conditions were met, it could automatically order a pregnancy test, ask the pharmacy system not to dispense the ordered medication, and notify the physician and nurse caring for the patient.

Like the interface engine modules, parsing modules of any type can be started, stopped, modified, created, or destroyed with no effect on any part of the process other than the database fields or alarms for which they are responsible and the queues from which they read their raw messages. As many parsing modules as necessary can be launched, running on a single machine or on multiple machines. In a high-volume situation, a single database field can be served by multiple parsing modules running in parallel.

Whenever a transaction modifies the contents of any field within the core database, the database automatically replicates that transaction to one or more duplicated databases. Duplicate databases provide redundancy in case of equipment failure, and they also permit load-balancing when a large number of client queries would otherwise tend to slow the responsiveness of the system. The core database is a logical entity, rather than a physical one, in the sense that any table or group of tables can share a physical machine, can enjoy a dedicated machine, or can be split across multiple machines. The combination of distributed data tables and redundant copies of each table allows the system to scale up to support large numbers of users and to tolerate a certain amount of hardware failure with no loss of service to the end-user.

End-user data access can be provided through any ODBC-compliant or SQL-compliant program running on a machine that is equipped with some means of user authentication. Independent authentication modules can be used to manage the socket-level connection for any commercial software, thus user authentication modules need not be integrated into the actual program that is used to query the data. Most end-users gain access to the data through one of several 'client data browser' programs that are self-installing,

self-updating, and that are heavily used both from the desktops of clinicians and administrators and in the shared clinical and administrative areas of the emergency department. Several copies are running on machines in the homes of physician-administrators, and are connected to the hospital by dial-up connection or by internet connections through a proxy firewall.

Client browsers use SQL or ODBC requests to obtain data. Their requests are received and processed by a secure transaction module that applies user-specific security rules to determine what browser components and functions may be enabled and what data elements may be requested. Once security rules are satisfied, the transaction module returns the requested data set to the client browser. The appearance and presentation of the information is determined by options that are set by each user and are saved as "named views" belonging to that user, thus the same data may be displayed in many different ways. Standard basic views allow single-click access to all clinical information for a single patient as well as facilitating filtering, sorting, and browsing through cohorts of patients that may be defined using any combination of the hundreds of data fields that exist within the system.

End-user data browsers are not the only client programs that query the data. Any authorized process can query and receive data, and a number of other independent processes do so on a scheduled basis. These processes use the data they receive to generate regularly scheduled reports, to send data to other systems (for example, to an outside billing company), and for a variety of other purposes.

Benefits of modularity and open standards within the model

Data acquisition occurs via a series of independent connections defined according to an open standard, so new connections can be added and the connection process controllers can be rewritten (or a new one purchased) at any time, without disturbing any of the other parts of the system. Inter-process communication occurs through open standard protocols – either by sockets connections or by SQL or ODBC queries – so any module can be stopped, started, twinned, modified, or upgraded without concern for any other module. The placement of incoming data into the appropriate field is driven by a system of scripted rules and procedures that are tightly coupled only to a specific field or group of fields, so new data elements and the scripts to populate them can be added at any time without affecting the remainder of the system. Data storage hardware and

software devices conform to an industry standard, so the system can be upgraded to take advantage of newer, faster or more powerful database programs and devices without changing any of the rest of the system. The database uses a standard security model and responds to a standard query language, so new connection-independent front-end programs can be added at any time without regard for how the data arrives in the database or how it is stored.

Where open standards are followed and a high degree of modularity is used, it is possible to mix together hardware and software from many different vendors to assemble a system that is both robust and agile. Open standards allow the database to be populated by many different types of data sources with no concern for which vendor has built the data input or capture system. Open standards allow the user to use the data in many different ways in front-end applications developed by different vendors.

Summary

Most hospitals and emergency departments cannot afford to scrap a large, highly integrated, expensive system and purchase a new one every three to five years, yet every proprietary system being sold today will be an impediment to progress at some point within that time period. In contrast, modern computer hardware, operating system and software solutions evolve and change very quickly. The distributed data-centric model of medical computing uses open standards and a high degree of modularity to permit the creation of a 'living' system that can evolve rapidly to meet the changing needs of the user and to integrate new software and hardware improvements as rapidly as they become available. With this model, the old question of "buy versus build" is replaced with the more modern decision to "buy AND build." This is not a theoretical model, but one that has been put into practice using a combination of custom and off-the-shelf modules at the Washington Hospital Center, where a successful production implementation is in daily use.

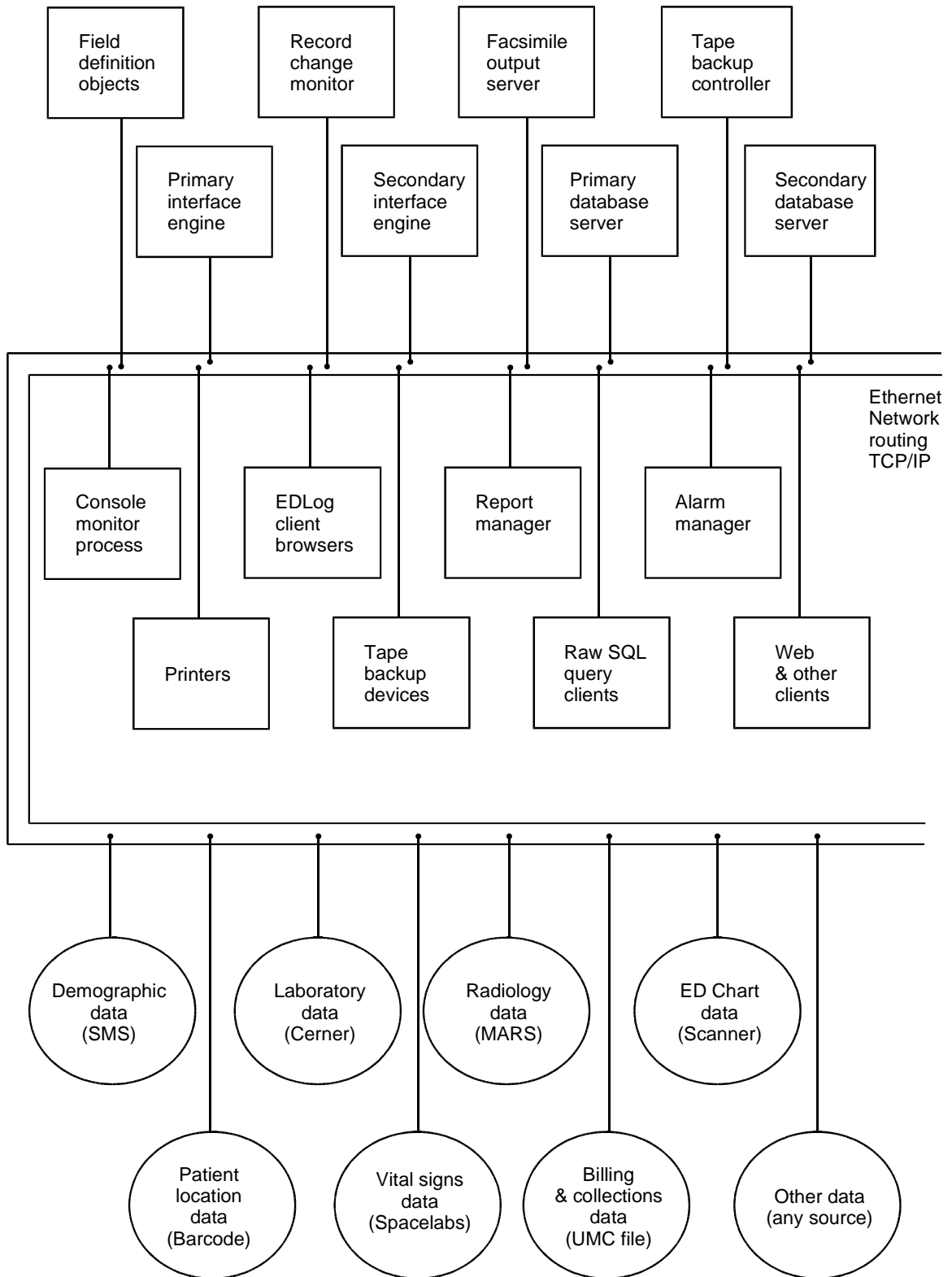


Figure 1
Every object communicates with every other using open standards