

5 Middleware

G. MICHAEL YOUNGBLOOD

The University of Texas at Arlington

5.1 INTRODUCTION

Developing systems and software for a smart environment is a daunting task. As with any large endeavor, there are many components that comprise modern smart environments. There is sensor hardware and software perceiving the environment, application software that interprets and reasons about that perception data, and effector control software acting upon the environment as well as many support systems. Most of these components are working with an operating system on hardware that makes computation possible with the goal of making the environment “smart.” Here lies the opportunity for software that can provide important services to facilitate rapid development, ease of integration, improved reliability, and increased scalability of systems that make smart environments possible. This software is commonly called middleware and this chapter is motivated by the search for the ideal smart environments middleware.

Middleware is connectivity software that joins applications through communication mechanisms creating transparency, scalability, and interoperability and lies between the software applications it assists and the platform it is based upon. Middleware typically resides in a layer built directly upon the operating system of the target hardware platform, but may be built upon other layers of middleware typically forming higher abstractions with each additional layer. Middleware is defined by the API (Application Programming Interface) it provides to applications that utilize it and the protocol(s) it supports [Ber96]. Middleware should be designed to reduce the complexities of the network, host operating system, and any available resource servers creating value in simplifying these for the applications using it and the developers who write them [Lin97]. Middleware should provide a cross-platform infrastructure that facilitates rapid development by providing services and features that would normally have to be developed if not provided by the operating system and a consistent and natural extension to the development of software applications. Overall, middleware should improve the desired characteristics of the target application and the developed system.

The goals of this chapter are to provide an understanding of the concept of middleware, present the desirable characteristics, and discuss its forms as well as the positives and negatives of each form. We will introduce some of the technologies in current use, develop an understanding of the importance of standards and where to find them, and present design issues to consider to assist in choosing the proper form and technology for a project. We hope to develop an understanding of the benefits of middleware and will present what current intelligent environments use for middleware that may provide some insight, ideas, and options for other smart environment projects. To accomplish these goals we will start by discussing some basic software architecture and the desirable characteristics of middleware that would enhance system designs, then the current forms and trends will be presented and discussed which will lead to a further discussion on middleware standards. Following that discussion, we will expand on design considerations before delving into what middleware systems are being utilized in current smart environment projects, and finally we will end with a discussion to summarize what we have covered.

This is an information chapter. Every smart environment project is slightly different, each with unique requirements, architecture, systems, and goals. We wish that we could give a definitive answer as to which, if any, middleware solution best fits a set of given project needs, but this would be a very large undertaking and the subject of a book in itself. Our goal in this chapter is to provide a working knowledge base to the smart environment designer, system architect, and developer so they may have the basic tools to begin an

examination of their system needs against possible middleware solutions. We are committed to the search for the ideal smart environments middleware, but we realize in advance that there is no silver bullet [Bro95] and that this will be a limited breadth-first search. This chapter is a survey of what is currently in common use, but is by no means comprehensive. The reader is strongly encouraged to use this chapter as a knowledge base from which to explore more deeply and broadly the world of middleware.

Before we get started, we will build an analogy that we can apply as we discover more about middleware. Since we are more than likely developing a smart environment, or at least thinking about it, let's use the analogy of a building structure—a house. The hardware platform will be represented by the foundation of the house and the operating system will be the general structure of the house (i.e., outside veneer, frame, walls, doors, windows, roof). In this house we will place the standard compliment of typical home appliances such as a telephone, television, stereo system, refrigerator, gas oven, microwave, telephone, and so forth. In setting up this home, everything seems in place. So what about the middleware? What would represent the middleware? We haven't introduced any extra technology in the home and this is not a smart environment as described, but there is a layer of non-software middleware.

Using our home analogy, we shall add a telephone to the home. The telephone can be used to communicate (among other things) with people on other telephones across the Earth. The telephone represents an application. The middleware is defined by its interface, the telephone wire (RJ-11), and its protocol, analog voice waveform. The middleware can be extended to the local coder-decoder (codec) where we can say exists the boundary for the operating system and networking (i.e., the digital telephone system). The middleware converts the analog signal to a digital signal handling a complex conversion so that the phone does not have to and supplies the signal and routing information to the telephone company infrastructure; conversely, the middleware will in full-duplex fashion decode digital signals into analog signals for playback through the phone. Thus, middleware greatly simplifies the application's required resources, provides a clear protocol, and a standard and natural interface (at this time in history humankind is very familiar with plugging in wired appliances). Many middleware products facilitate the client-server relationship where there is a consumer and a producer, or the peer-to-peer relationship where there is an exchange of information/services. This home analogy extends to all basic services. For example, the gas stove is the client application, the gas company is the server, and the middleware is the gas piping where the interface is the gas adapter fittings and the protocol is natural gas (mostly methane). Middleware provides an interface and a protocol, but what other characteristics can it possess?

5.2 NEEDS, WANTS AND DESIRES

There are possibly an infinite number of approaches and paradigm combinations to explore when building smart environments, but we shall use a basic perceive-reason-act AI (Artificial Intelligence) approach [RN95] as a building block to identify a typical set of software needs, add in some wants, and define desirable characteristics for middleware we might consider.

Start with a basic plain house. In this house we are going to place sensors (e.g., temperature, humidity, light)—these will be our senses or perceptory input through which we *perceive*. Logically each sensor or group of sensors is represented in a computer by a simple application that allows a specific sensor to be read, produce a continuous reading, perform calibration, and other appropriate operations. Next we add some software AI applications (more than one) to *reason* about this environment based on perception in order to provide an action to change the system state—hopefully for the better. Lets arrange these so that there are three AI applications where two receive sensor data as inputs, perform some independent reasoning on this data and then they both pass abstract data to a single decision-making application that will decide whether or not to take an action by doing nothing or passing information onto a controller application. Finally we add some controls over the environment (e.g., power control, light control, water flow control) as mentioned previously—these are our actions that we can take to affect our environment through which we *act*. Figure 5.1 provides a graphical representation of this design.

The basics are put forth, but there are still many unanswered questions about this design and how it should work. We are working to build an architecture comprised of several different components that when working together, each component performing as designed, comprise a complete perceive-reason-act based system to control an environment. However, for this chapter we are not concerned with the inner workings

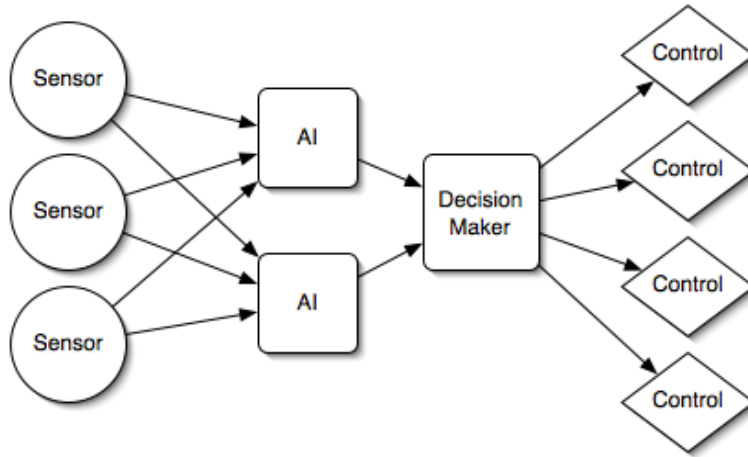


Fig. 5.1 Basic Perceive-Reason-Act Design

of each component, but rather by their communication requirements with each other and features that can improve this system’s functionality preferably with little extra work on the developer’s behalf.

The needs of this system are first defined by the need for application interoperability—the ability for two or more entities to exchange and interpret information from each other [IoEEE90]. The sensors need to provide information to the input AI applications, the AI applications need to provide information to the Decision-Maker, and the Decision-maker needs to pass information to a controller. Just the act of communication brings out further needs. We want to ensure that when communication occurs the messages have a delivery guarantee, so the mechanism must be reliable. The communication process should minimize resource consumption and time of delivery, so it must be efficient. These applications may provide a large amount of data traffic in their intercommunication, so the middleware should have a high level of throughput. The middleware should hopefully not be the bottleneck in any system unless that is inherent to the problem—in which case it should provide a clear advantage over other solutions. In addition to our basic interoperability need, there should not be a restriction that all of these applications reside on a single piece of hardware, middleware should allow this communication to occur among these applications through a network with the possibility of all components being on the same machine, all on different machines, or a combination (i.e., distributed computing support). These machines may be in the same area or geographically distributed. Reliable, efficient, and distributed communication is our basic need.

In addition to these needs we would also want middleware that provides for future extensions and contingencies. If we decide to add more sensors or controllers to our system, we would want the system to scale up appropriately and easily to accommodate larger sized systems. Furthermore, if we decide to replace an existing application in the system the middleware should allow for hot-swappable components in which the system does not have to be shutdown in order to replace an application component. If important data is transmitted between applications then there should be a secure means of communication to protect the sensitive data. While the middleware provides these features, we want such a system to be highly available, always running and accessible to applications in need of the middleware services, and to be fault-tolerant, continuing operations despite failures in hardware or software components [IoEEE90]. We certainly do not want a fragile system or one prone to complete failure if one of the distributed component applications fail. The addition of scalability, hot-swapability, and security in a highly available, fault-tolerant middleware solution improves the value that a middleware solution can provide.

Given the mentioned needs and wants, lets add some additional desirable characteristics. Thinking about the developers—they want simplicity and power. Middleware APIs should provide simple (low complexity) but powerful interfaces with a high level of transparency—interfaces that follow the same form and function as other integral parts of the implementation (e.g., in C++ a transparent API may simply just add additional classes which when instantiated appear as any other object in the system and with which standard means of use apply such as using the stream insertion operator to pass information to that ob-

ject). Utilizing a middleware solution should feel like a natural and seamless extension of the development environment for the implementer and not present an overly complex system that removes focus from the actual purpose of the system under development. The middleware interface should offer flexibility to allow easy modification of software components so they can interact in different environments. Middleware that provides means for developing well defined interfaces while maintaining a separation from the implementation greatly aid in maintaining many programming paradigms, as well as aiding flexibility, and provides interface maintainability and reusability. Maintainability is important to development and for the future life of a system—it must have the ability to be modified in order to correct errors, make improvements, or be adapted to environmental changes [IoEEE90]. Reusability is also important to developers who like to take common solutions to problems and use them in different systems. Middleware should make reusability easier and perhaps more intuitive. While being maintainable and reusable, that reusability would be greatly enhanced by a high degree of portability. Middleware should be available on many platforms and provide a consistent interface across them allowing the same development code to be used in multiple environments. It is desirable for middleware solutions to have low complexity and transparent interfaces while providing flexibility, maintainability, reusability, and portability for developers to more easily adopt, integrate, and utilize these solutions to improve their systems and applications.

This is just an overview of the process that system architects and developers should go through when designing smart environments and making middleware considerations. There is much to consider in designing and building these systems. Every uniquely designed system will have a set of unique requirements. System architects should carefully evaluate the system requirements when evaluating a fit for middleware solutions. At a minimum, the benefits that middleware should provide are improved functionality, reliability, and development cycles through simple, natural interfaces and common services. System needs, wants, and desires are creating the demand for middleware that provides efficient, reliable, interoperable, flexible, portable, secure, scalable, maintainable, reusable, adaptable, and transparent simple services. For the interested reader, the Software Engineering Institute at Carnegie-Mellon University provides more information concerning software characteristics and design [Ins03].

5.3 IN-BETWEEN THE OPERATING SYSTEM AND THE APPLICATION

Now that we have a better understanding of what middleware is and what we are looking for in a middleware solution, we should explore these systems that reside in-between the operating system and the using applications. In this section we will discuss some architecture basics in preparation to talk about the forms of middleware and follow with frameworks, middleware services, and ubicomp initiatives.

5.3.1 Architecture Basics

Middleware is mostly about communication. Before we begin discussions on the forms of middleware, we shall start with a review of some basic communication paradigms. Computing began with close human-computer interfaces requiring re-wiring the computer, using punch-cards or paper tape, and even flipping switches before the field advanced to teletype interfaces. Eventually computing moved to the mainframe architecture which consists of a central mainframe computer where all computation is performed and users interacting with the system through terminals which simply capture keystrokes and relay the information to the mainframe. Over time and with the advent of personal computers (PCs), computing moved towards a file sharing architecture utilizing these new PCs to download files from shared locations to the local desktop where user jobs were run. In the 1990s when more and more people began using PCs, the PC local area networks (LANs) became too overloaded for this paradigm [Ede94].

The limitations of the file sharing architecture gave birth to the client-server architecture. The file server was replaced by a database server which responds to queries by returning only requested information instead of whole files. This reduced network traffic load and made way for incorporating graphical-based clients to provide a consistent desktop experience [Ede94, Sch95]. Client-server architectures are typically implemented in two or three tier architectures.

Two-tier client-server architectures are defined by a client layer and a server layer as shown in Figure 5.2. This design assigns the system interface (e.g., session management, display interface, input/output) to

the client and the database management system (DBMS) to the server. There is an additional component of the system that is shared between the two tiers—processing management that manages the initiation, conduct, and publication of the results from work to be performed. This migration to a more specialized system of services that perform specific work and clients that request work utilizes resources more effectively and efficiently and allows for greater specialization of the server to perform work and of the client to be more tailored to the capabilities of the client platform. This provides scalability and flexibility over the file sharing architecture and performs well in light traffic conditions [Ins03].

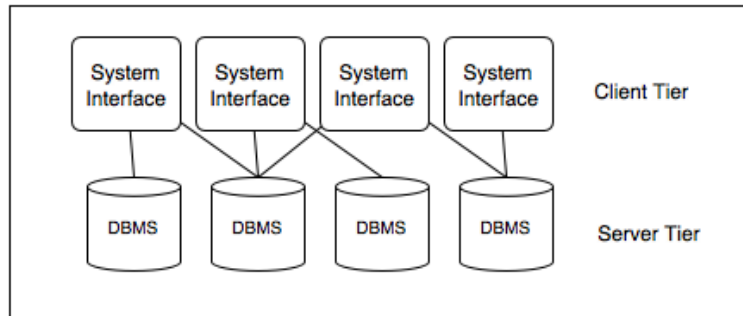


Fig. 5.2 Two-tier Client-Server Architecture [Ins03]

The three tier client-server architecture as shown in Figure 5.3 was designed to compensate for the two-tier architecture’s shortcomings. A middle tier is added between the system interface and the DBMS. This tier takes over the process management by providing logic and rules to control job processing and adds features such as queueing of messages. The addition of this layer and features it provides increases the number of clients the system can handle over a two-tier (increased performance) and improves the flexibility, maintainability, reusability, and scalability [Ins03]. This architecture has become the basis for many middleware solutions.

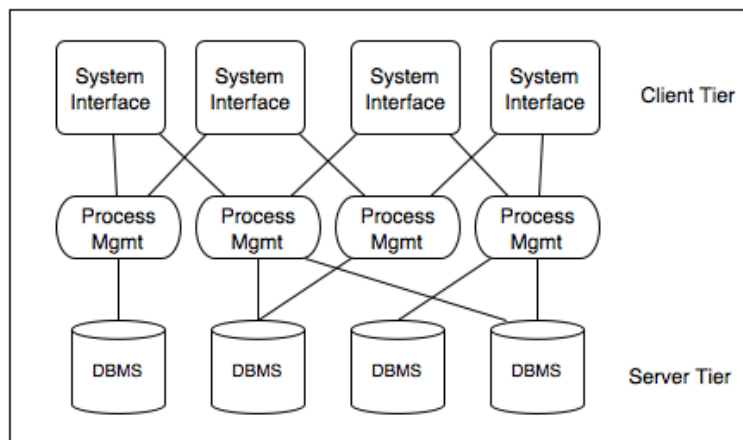


Fig. 5.3 Three-tier Client-Server Architecture [Ins03]

Given the background information on communication architectures, we are now ready to examine how middleware has grown from this architectural beginning as our worldwide information systems grow in size and complexity.

5.3.2 Forms of Middleware

Middleware comes in many forms, but the forms presented here are the most typical architectures being utilized presently. The number of middleware products is quite staggering and many involved in this field

have slightly different views on the role, terminology, and composition of these systems. This is just the nature of the area of middleware and is an artifact of how rapidly it advances. After all, when programmers only wrote machine code for specific platforms and those were the only programs that ran on a machine (one at a time) along came the idea of a novel piece of middleware, the operating system, which sat between the hardware and the application to allow programmers to write code for an operating system instead of a specific hardware platform. In the next sections we will present the basic forms in high-level categories of middleware. In each section we will present an overview of the technology, discuss its strengths and weaknesses, and then list some implementation examples for the reader's reference.

5.3.2.1 Transaction Processing Transaction Processing (TP) monitor technology is involved with creating a middle tier of processing routines between a system that provides transaction based services (e.g., financial, sales) and clients making those transactions. Many TP systems utilize the two-phase commit protocol [LS79] which can be demanding on a high use system. Introducing a layer of processing routines to manage the transactions more loosely with the clients in a stateless manner while being statefully coupled to the transaction server enables orders of magnitude more clients while satisfying the rigid transaction protocols. Figure 5.4 illustrates this technology.

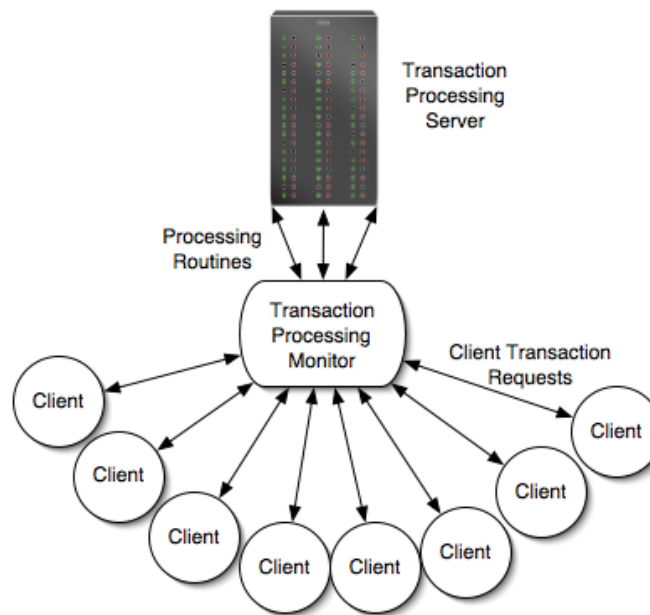


Fig. 5.4 Transaction Processing Monitor Technology [Ins03]

TP monitor technology usually includes the abilities to handle client disconnect and reconnects seamlessly, restart failed processes to maintain availability, marshal data from various types of clients into a consistent form for transaction processing, perform load balancing, and provide consistent logic across clients.

The advantages of this type of middleware includes a level of independence of the key layers—the client is independent and can be customized for specific platforms, the transaction processing monitor is independent and can take any clients that communicate with the proper transaction protocols and can communicate with a transaction server over a defined transaction protocol, and the transaction server and underlying database are transparent to the TP monitor. This system is efficient and reliable at creating a flexible transaction-based system, and they are very mature and well tested in large scale deployments servicing millions of transactions per day.

The disadvantages of this type of middleware are the limited scalability due to each client incurring a small amount of system overhead on the TP monitor and, due to being an older form of middleware, most of these systems are written-in and for lower-level programming languages.

Examples of TP monitoring technology implementations are IBM's CICS TP monitor and BEA TUXEDO.

5.3.2.2 Message-Oriented Message-Oriented Middleware (MOM) is a layer of software that resides upon the operating system and provides communication services between applications either on the same machine or on other machines on possibly different platforms. These messages are generally asynchronous, but could be synchronous (sometimes, if desired), and the middleware usually provides message queuing and persistence for those times that the message recipient is unavailable and delivery is postponed until their availability. MOM establishes a peer-to-peer level of connectivity between applications breaking down the strict client-server relationship that sometimes forces master-slave relationships. The messages can follow agreed protocols and may include structure and form freeing the developer from cross-platform issues such as endianness [Ste95]. Figure 5.5 illustrates a typical MOM system.

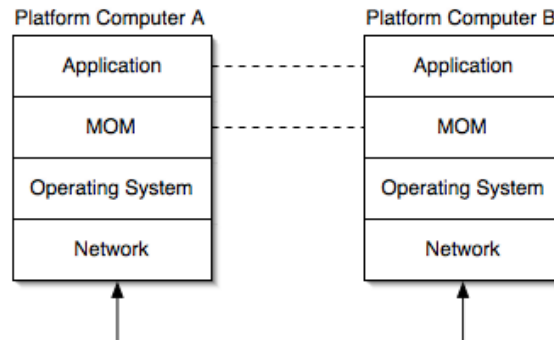


Fig. 5.5 Message-Oriented Middleware [Ins03]

MOM provides the advantages of abstracting away cross-platform and local communication complexities, increased interoperability between applications through communication, cross-platform portability for messaging, and increased flexibility. MOM is a good-fit for event-driven applications and peer-to-peer communications. Asynchronous communications allow continued operations of the system instead of waiting on message delivery and response. MOM has been around since the 1980s and is a mature technology.

The disadvantages in MOM are that with asynchronous communications it is easy for a client to overload the network and server that is not processing the messages quickly. MOM must run on every platform in a system and if there is not an implementation for a platform then the system will not work. There may also be limitations on protocol support with less popular platforms.

Examples of MOM systems are Oracle Advanced Queueing, Arjuna Messaging, IBM MQSeries, and Microsoft MSMQ.

5.3.2.3 Object-Oriented Object-oriented middleware or Object-Request Brokers (ORB) are middleware technologies that extend the object-oriented development and communication paradigms to middleware. ORBs support interface definition languages, object communication mechanisms, and object activation and location mechanisms. ORBs facilitate locating objects and establishing communication with those objects in a manner similar to MOM technology. ORB interfaces and object communications usually appear as transparent extensions to the object-oriented development environment. Objects that may be remotely located seem local to the objects communicating with them. Figure 5.6 illustrates an ORB system.

The advantages of ORB technology are those of MOM with the addition of the interface usually being a natural extension to the object-oriented development environment. Interface description languages employed by ORB systems provide a clear mechanism for providing contractual interfaces between objects, provide for rapid integration, and preserve implementation and interface separation.

The disadvantages of ORB technology include the issue that different ORBs provide different levels of service, support ranges of platforms, and often only support certain OO languages. Finding the ORB to meet all of the software needs requires careful examination of the available ORB implementations [AEGO96].

Examples of object-oriented middleware include OMG's CORBA, Microsoft's COM/DCOM, and Sun's Java RMI.

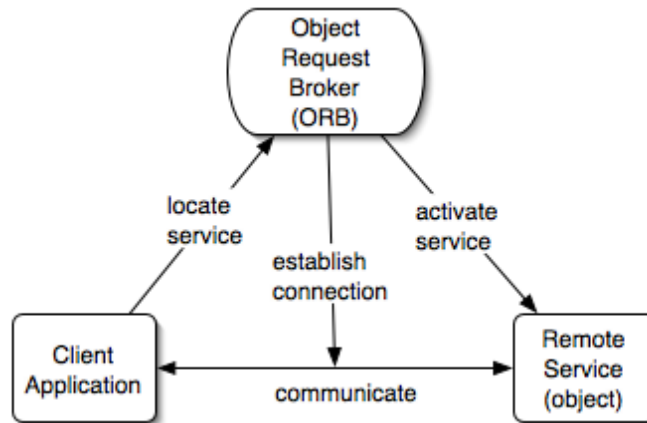


Fig. 5.6 Object Request Broker [Ins03]

5.3.2.4 Database Database connectivity issues can be complicated. Middleware products have been developed to provide API access to standard database interfaces in an effort to answer the call for simplified and robust database connectivity. This layer takes the form of development connectivity tools and language extensions that facilitate the intricacies of application to database communications.

The advantages of this middleware comes from standard and simplified database connectivity, but the disadvantages are that many solutions are not cross-platform, may not support advanced database features (especially proprietary ones), and may provide blocking, synchronous connections.

Examples of database middleware include Microsoft's ODBC, Sun's JDBC, and Rogue Wave Software Inc.'s DBTools.h++ [Lin97].

5.3.2.5 Remote Procedure Call A more discreet form of middleware lies in Remote Procedure Call (RPC) technology. RPCs are stubs embedded into the client-server applications at compile-time. The stubs facilitate procedure calls between the client and server. Figure 5.7 shows the structure of RPC middleware [Ste95].

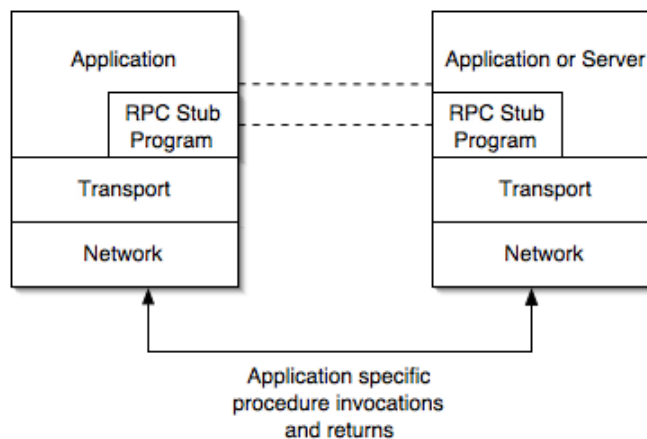


Fig. 5.7 Remote Procedure Calls [Ste95]

The RPC advantages include providing a level of consistency in procedure calls both locally and remotely, a natural extension for remotely calling server functions, and network transparency of the client-server locations.

The disadvantages of RPCs stem from the fact that the majority of RPCs are implemented using synchronous communications forcing a call-wait scenario that can cause blockages in the flow of an application.

Even in cases where RPCs include asynchronous mechanisms they add significant complexity to the development [Rao95].

Synchronous RPC can be an advantage to keeping a system simple and preventing network overloading, but make the technology a poor choice for object-oriented systems or peer-to-peer implementations.

The Open Group's Distributed Computing Environment is an example of an RPC system [TOG03, Blo92].

5.3.2.6 Web Services Web services have been popular due to bridging the interface gap between applications that are often hidden by network security features such as firewalls. Leveraging the basics of technologies that have created the ubiquitous World Wide Web, a web service is simply a network accessible interface to an application using web-based protocols (e.g., HTTP, XML) [STK02]. Typically, the interface is provided by ASCII text-based XML (eXtensible Markup Language) communicated between applications via HTTP (Hyper-Text Transport Protocol). XML merely specifies a format language though; the true definition of the interface is molded by other emerging technologies such as SOAP (Simple Object Access Protocol) that adds object-oriented mechanisms to web services and WSDL (Web Services Definition Language) which provides communication descriptions between web service endpoints. Figure 5.8 illustrates how web services work.

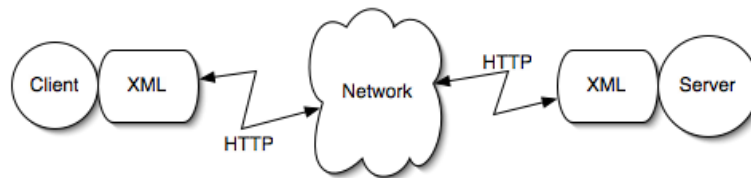


Fig. 5.8 Web Service Structure Example

The advantages of web services are that due to the ubiquitous nature of web servers and browser APIs of which they take advantage, there is a wealth of software to facilitate infrastructure and implementation. Communication over an ASCII-based message protocol improves troubleshooting due to complete message transparency and almost human-readable message traffic. Since most corporate security systems already allow web traffic into and out of the enterprise, communication barriers for web services are few. Network administrators can easily ensure web traffic channels are available.

The exploitation of being able to easily pass XML traffic through HTTP traffic channels on the network has led to potential security holes and the implementation of network equipment that can filter by traffic content. Another weakness of web services is that the conversion of data structures into XML and the subsequent parsing is computationally expensive compared to more native binary data formats; thus, web services have a tendency to be slow. Describing data in XML also greatly expands the data structure size utilizing more bandwidth for communications.

Examples of web service building tools and infrastructure support include Apple Computer's WebObjects, IBM's Websphere, Microsoft's .NET Framework, and Sun's Open Net Environment.

5.3.2.7 Agent-Oriented Coming from agent research being conducted in universities around the world is the basis for the emergence for a new type of middleware—one that is centered around realizing software agents. Agents are autonomous, intelligent software entities that have the ability to perceive their environment, reason about what they observe, and act to affect changes upon that environment to accomplish their goals [RN95, ISAG03]. Agent technologies are being used to solve financial management, military logistics, personal information management, and other problems [ISAG03]. Since agents need to perceive and act upon their environment, communicate with outside entities and other agents, and may have platform and network mobility middleware components, multi-agents systems (MAS) have emerged to facilitate agent needs. These packages typically appear more as frameworks (to be discussed later) and provide means for inter-agent communication, load-balancing, and mobility (moving agents between machines), but some systems utilize extensions to MOM and object-oriented middleware systems creating higher levels of middleware solutions. Agent middleware may provide one of the areas of growth in this field as this paradigm expands.

Examples of agent frameworks are HIVE [MGR⁺99] and CMU's RETSINA [SDP⁺96].

5.3.3 Frameworks

Not to be confused with the middleware forms we have discussed previously, but worth mentioning is the area of frameworks. Frameworks are targeted at specific domains and consist of software environments that provide an API, a user interface, and tools for application development and system management [Ber96]. Frameworks may provide their own private middleware services and may utilize other common middleware services. For specific domains, frameworks may provide better solutions than general middleware and may be accurately called middleware themselves.

Examples of some frameworks include Lotus Notes, Microsoft Office, Transarc's Encina, Cognos, and HP's OpenView [Ber96].

5.3.4 Middleware Services

This may be an area of definitional debate as some people consider the forms of the middleware described previously to be middleware services and middleware itself to be those services and/or frameworks [Ber96]. We choose to refer to distributed system services as middleware under the premise that good middleware is never seen (transparent) and tends to be useless unless used by an application; whereas, frameworks usually provide a user interface and environment and are therefore visible to the user. We will just refer to them as frameworks for now and *middleware services* shall be defined as service components that provide value added work to other middleware clients. Such middleware services include naming services, directory services, interface repositories, and data libraries.

As middleware evolves, abstracting commonly implemented components into the middleware adds value and functionality that reduces the development burden and creates portable, common solutions to commonly occurring problems. We will not discuss all of the middleware services available today, but rather let us focus on the trend of services provided.

When distributing components across the network it is often difficult to locate those components, and more importantly being to locate useful components. Naming services create central repository locations for performing name lookups from common names to resource names which provide location and connection information. When a component is not sure of another component's name it seeks to connect to, but knows what type of component it is, a trading service can broker the connection by providing matching names based on desired characteristics to requesting clients—they are also known as discovery services [HV99].

Services may even provide modified communication schemes such as an event service that can provide a more decoupled publish-subscribe communication method augmenting or replacing point-to-point communications [HV99]. Services can add value and flexibility to middleware.

5.3.5 Ubicomp Initiatives

In a world soon to be dominated by ubiquitous computing (ubicomp), several initiatives have emerged to tame the potential madness. In the current state of technology, when a user purchases new computer peripheral devices (e.g., printers, scanners, PDAs) the user has to load in drivers and configure items before being able to fully utilize those new devices. As more and more items have embedded microchips placed in them and gain features that allow them to become managed or peer computing peripherals, the user-required configuration scheme may become unmanageable, but technologies such as ZEROCONF [Che03] and Universal Plug and Play (UPnP) [UPn03] seek to alleviate the problem through zero configuration technologies that allow devices to self integrate and configure in their local environments. These technologies may become part of the middleware infrastructure and are already embedded into the operating systems of Apple (Rendezvous branded ZEROCONF) and Microsoft (UPnP). Zero configuration technologies are poised to replace name and service discovery middleware services.

5.4 STANDARDS

A common mistake for the new designer is to think they are the first to ever solve a problem or that their solution is better than anyone else's. This type of thinking leads to islands of technology and repetition of

research and effort. By introducing some standards and sources, we hope that readers will utilize existing standards and strive to improve them to allow scientific and engineering progress to advance and to avoid repetition of work. To this end, we will start by discussing some of the middleware standards in common use and introduce the organizations leading the standardization effort. Following this will be a discussion on protocol standards.

5.4.1 Middleware Standards and Implementations

The growth of middleware and its core communication-based underpinnings has caused an inherent need to create commonality in functionality, interface, and protocol so that even though different implementers will provide different solutions, they should still be able to interoperate. Some standards are led by industry independent organizations that act as consortiums for collaboration and compromise on the standards producing open, well-defined standards and others coming from proprietary corporations become *de facto* standards due to large market share and heavy corporate influence.

5.4.1.1 COM/DCOM An example of a *de facto* standard is the Component Object Model (COM), Distributed COM (DCOM) middleware from Microsoft, Inc. What was originally a Windows technology but has since migrated to other platforms, COM provides a model for interface definition through the IDL (Interface Description Language) and a protocol for communication between objects. DCOM extends this to components across the network.

5.4.1.2 CORBA A more open standard has been defined by the Object Management Group [OMG03] in CORBA (Common Object Request Broker Architecture). CORBA is part of a larger standard called the Object Management Architecture (OMA) which defines how application interfaces, domain interfaces, object request brokers, and object services interact. CORBA uses an IDL-to-programming language mapping for the large number of object-oriented languages supported. Skeleton and stub code are generated and compiled into developer code which interact with an Object Request Broker (ORB) on each machine. The ORBs talk to each other in a distributed system through the network using an inter-ORB protocol (e.g. IIOP, the Internet Inter-ORB Protocol) to facilitate a complete communication infrastructure. Services such as naming, trading, and event as well as IDL repositories and dynamic capabilities make CORBA a powerful and widely used middleware standard.

5.4.1.3 DCE Another open standard is the Open Group's Distributed Computing Environment (DCE) [TOG03]. DCE is composed of a set of integrated system service specifications which are operating system, network, and platform independent. DCE provides fundamental distributed services such as remote procedure calls (RPC), security, directory, time, and threads as well as data-sharing services such as a distributed file system and diskless workstation support [Ins03]. DCE has been very popular and forms the base of many other middleware layers. Recent extensions to improve object and web compatibility create continued use and growth opportunities for DCE.

5.4.1.4 Java Middleware Technologies Sun Microsystems' Java language was originally designed for embedded devices and found quick adoption during the growing use of the Internet by providing a mechanism to distribute operational code over the network via applets. Java was also always capable of creating applications, and due to its portability has been growing in usage. JIT (Just In-Time) compilers and optimizers have improved the interpreted language's speed and efficiency.

The Java platform hosts a number of middleware initiatives and support. Java supports CORBA as part of its platform and from many vendor products. Java also offers the proprietary Java Remote Method Invocation (RMI) mechanisms that provide a CORBA-like object-oriented middleware layer for distributed inter Java-object communication. In cooperation with industry partners, Sun has developed the Java Message Service (JMS) API as part of the Java 2 Enterprise Edition to provide MOM to enterprise Java systems. Java also participates in the web services realm by offering the Java Web Services Developer Pack (Java WSDP) to facilitate easy integration of web services into Java applications. Java Servlet technology and Java Server Pages (JSP) provide the means to extend web server functionality and provide dynamic content to support web service serving applications. And if that was not enough to offer, Java Jini technology is being deployed

to provide adaptive network-centric applications and services from dynamic networked components through middleware that offers code mobility, open protocols, fault tolerance, and ease of integration [SM03]. The Java platform is poised to offer it all as long as the developer adopts programming in Java and does not mind being completely dependent on technologies from a single source.

5.4.2 Web Services Standards

The World Wide Web Consortium (W3C), fueled by strong industry focus on web services, is leading the way to provide standards for web interoperable technologies—specifications, guidelines, software, and tools [WWWC03]. Specifications for HTML, HTTP, XML, SOAP/XMLP, WSDL, and other key technologies can be found freely available from their website.

In addition to the W3C there is also the Organization for the Advancement of Structured Information Standards (OASIS). OASIS provides another membership-driven standards body that is currently offering specifications for DocBook (a documentation standard), Directory Services Markup Language (DSML), ebXML (eBusiness XML), Security Assertion Markup Language (SAML), and the Universal Description, Discovery and Integration of Web Services (UDDI) standard among others [OAS03a]. UDDI is bringing trading services to web services [OAS03b].

As mentioned in the last section, there are also many vendors providing development tools, APIs, and services that can facilitate faster web service development. As this area matures, many may fall out of use and dominant forms may become *de facto* standards.

5.4.3 Databases

Databases provide an important role in modern systems as storage caches for often vast amount of information that is the key component to processing systems. Structured Query Language (SQL), created by Oracle based on SEQUEL by IBM in the mid-1970s, has become the *de facto* database communication language even though many vendors provide non-standard extensions [Web03]. Open DataBase Connectivity (ODBC) developed by Microsoft provides a middleware database driver to facilitate standard database communications between applications and a database. Many database vendors produce ODBC-compliant databases. Sun provides ODBC for Java in the form of JDBC.

5.4.4 Protocols

There are protocols for just about everything one can imagine in computing. This chapter has already mentioned many protocols and sources for information, but in addition to just communication negotiation protocols there are specifications for many of the intricate information transactions that take place between applications and/or systems. A good source for protocol information is to start with the Internet Engineering Task Force's (IETF) Request For Comments (RFC) website [IET03] where authors submit protocol specifications for comments and opinions or the Internet Assigned Numbers Authority (IANA) website [IAN03] where port numbers are assigned to services that can be traced to the protocols used by those services. As stated in the web services section, W3C [WWWC03] and OASIS [OAS03a] provide protocol standards. The Open Group [TOG03] and Object Management Group [OMG03] provide them as well. Standards bodies such as the ISO [IOFS03] and ANSI [ANSI03] may provide internationally or United States approved standards. For mobile communications there is the Open Mobile Alliance (OMA) [OMA03] and more broadly for telecommunications there is the International Telecommunication Union Telecommunication Standardization Sector (ITU-T) [ITU03].

Utilizing standards and providing feedback and participation to the using community of those standards helps them meet their intended goals and become mature technologies. Standards provide contracts for interoperability and allow powerful systems to be built.

5.5 DESIGN CONSIDERATIONS

In section 5.2 we discussed our needs, wants, and desires as architects and developers of smart environment systems. In section 5.3 and 5.4 we covered middleware basics, forms, implementations, and standards. Now lets revisit those design discussions.

One may have determined that they don't even need middleware for what they are going to be working on. That argument has been made many times to the detriment of all that could have used someone's work had it been more easily integrated into theirs. Researchers tend to be the worst offenders of this since they often prefer to work in isolation and are often prone to project tunnel vision. Take it into consideration that if there is even a remote possibility others may benefit from using a system, then they should (please) make it easy integrate. Middleware can help and this is why middleware is important.

Middleware should complement a project, making it easier to design, develop, and maintain. It should provide a level of interoperability, reliability, efficiency, and throughput handling to applications using it. It should also afford security, scalability, adaptability, dynamism, availability, and fault tolerance. For the future life of the application middleware should also allow flexibility, portability, maintainability, and reusability. Designers should know what they are looking for, what the limitations and requirements are and choose middleware that best fits the goals of the application, but should also be careful.

Caveat emptor! Middleware can also be wrought with pitfalls. Many middleware solutions can bring a large amount of infrastructure to a project. This may incur more system administration and problems than benefit. Many solutions are completely proprietary to a single vendor which place a large risk on a project if that vendor goes out of business or decides to discontinue the product. Even with standardized middleware, many vendors extend the standards to include their own proprietary extensions which may add value but will give the project a dependence on that particular vendor's solution (the same pitfall as a completely proprietary solution applies). Even if they are very tempting and offer an improvement, try to avoid using proprietary API extensions. As an alternative, open source projects often seem tempting and may provide some good middleware solutions (e.g., the fastest CORBA implementation, omniORB, is an open source project [Gri03]), but many open source projects do not last long as development is on a volunteer basis and many authors move on to paying work. An open source dependence may lead to middleware maturity issues which may drag the development team to working on the middleware instead of the target project. Open source licensing may make it difficult to sell a product for profit or may require that source code be distributed. However, open source projects may provide an excellent immediate solution and if an organization is willing to contribute back to the middleware project, may provide a more long-term solution.

Middleware is defined by its API and the protocols it uses [Ber96]. We mention that to stay standard avoid proprietary API extensions, but there is another concern. Protocols can change or even have dialects—more vendor extensions. The same rules should apply for protocols as APIs—do not use the proprietary protocol extensions, stay with the standard specified protocol. This is not to say that these extensions are evil or that they do not add value; some proprietary extensions even become part of the standard (e.g., Netscapisms that were added to the HTML specification). Middleware should grow with the standards they are based on and implement the newest features of their design and protocol specifications. Users of proprietary extensions need to consider the benefits of use over the retooling of the software if the middleware vendor were changed or if the middleware solution ceases to exist. They must also consider the impact on interoperability with other applications.

As projects utilize more and more middleware products there is a dependence shift from the operating system and platform services to the middleware and its services. Projects should look for middleware that provides for their needs, wants, and desires, but also follows an open standard that has many vendor (both commercial and open source) implementations and multiple platform support. Focus on what the project really needs for a middleware solution and what can best provide those needs.

Going back to the basic perceive-reason-act system we presented in section 5.2 Figure 5.1, we can apply some of our acquired knowledge from the previous sections to the design. This is just one possible design solution and not necessarily the best. Our purpose is just to illustrate the process of middleware selection. The needs of the system, as stated previously, are one of basic reliable, efficient, and distributed communication between the components. We will forego our wants and desires for now and leave that as an exercise for the reader to satisfy. Based on the basic need we have several options: a MOM solution, an Object-

oriented solution, a RPC solution, or a Web Services solution. There certainly is not an obvious transaction processing, database, or agent-oriented need. Given what we know, any of the solution options is viable, but if we add that we will be developing on the Linux platform using C++ on a closed network we can narrow our solution choices. These additional system requirements lead us eliminate the Web Services and MOM solution because XML over HTTP is too communication and processing inefficient and MOM APIs are programming inefficient with C++. Web Services could also be eliminated due to there not being a need to bridge across the Internet. RPC would be a good choice if we were using Java because of the built-in RPC mechanisms, it is also easy using C++ on Linux, but an Object-oriented solution fits better with C++. We now choose CORBA as our Object-oriented middleware solution because we are using Linux, if we were using Microsoft Windows we may have chosen COM/DCOM. CORBA IDL will provide a good environment for defining our own interfaces between the custom system components and has the benefit of not locking us into C++ if we should decide to replace a component with a Java, Python, or other CORBA-supported language component. CORBA provides a reliable, efficient, and distributed communication middleware system that fulfills our basic needs. CORBA is also object-oriented and available on Linux—meeting our development language and OS requirements. We could further delve into CORBA ORB selection and so forth, but it should be obvious to the reader that proper selection of middleware is merely a function of knowing your all of your system requirements and matching them with the middleware solution that best satisfies them. It should be noted that in some cases it may take several middleware solutions to meet a single system’s needs.

5.6 MIDDLEWARE FOR SMART ENVIRONMENTS

Creating new technologies and solving the outstanding research questions associated with smart environments presents a need for new viable middleware solutions. The unique real-time, sensory, control, and data flow issues presented by these environments can be a significant hurdle and a recognized impasse to accomplishing work in this area [Coe98]. To provide possible solutions to allow work to continue in their focused direction, we present a brief survey of the present and recent past middleware solutions over various smart environment projects. The projects are presented in alphabetical order by the encompassing project name. We strongly encourage the reader to further investigate these and other projects to learn from other project successes and failures.

5.6.1 AIRE, MIT Artificial Intelligence Lab

Out of the MIT Artificial Intelligence Lab comes the AIRE (Agent-based Intelligent Reactive Environments) group. The AIRE group is engaged in research involving pervasive computing designs and people-centric applications and have constructed “AIRE spaces” in the forms of an intelligent conference room, intelligent workspaces, kiosks, and “oxgenated” offices. To assist in their research and to integrate their research technologies, they have developed middleware called Meatglue and an extension, Hyperglue [Gro03].

Metagluе is an extension to the Java programming language to allow the creation of intelligent environment controlling software agents. It provides *linguistic primitives* that facilitate the interconnection and management of a large number of disparate components (hardware and software), real-time operations, agent management, resource allocation, dynamic configuration, dynamic components, and state capture mechanisms. Metagluе provides an inherited *agent* class to Java objects, a post-compiler to generate new Metagluе agents from Java-compiled classes, and a *Metagluе Virtual Machine* infrastructure to be run on all supporting computing equipment. The Metagluе authors claim that the infrastructure creates a negligible amount of overhead and is more efficient than CORBA or KQML (Knowledge Query and Manipulation Language) because it provides both communication and control with a lighter-weight solution [CPW⁺99].

Hyperglue extends Metagluе by providing a society communication model and discovery system for Metagluе agents in a new infrastructure layer. Metagluе allows for agents to be segmented into small groups called *societies*. Hyperglue provides a communication layer for societies to communicate with each other and to find needed service providing societies through resource managers and society ambassadors [PLQ⁺03].

5.6.2 The Aware Home, Georgia Tech

The Aware Home Research Initiative (AWRI) at the Georgia Institute of Technology is conducting research looking to answer the question, “Is it possible to create a home environment that is aware of its occupants’ whereabouts and activities? [AHR03]” Although not specifically middleware systems, they have conducted research developing two toolkits, INCA and the Context Toolkit, as well as a location service that may be of interest to system designers and integrators. INCA is the Infrastructure for Capture and Access, and it provides the means for creating systems that capture life experience details and preserve them for future access. INCA provides capture, storage, format-conversion, and access support [TA02]. The Context Toolkit provides abstractions and support for context-aware development. This Java toolkit consists of widgets as an encapsulation object, aggregators to create meta-widgets, and interpreters to translate low-level context information into higher levels all communicating via XML over HTTP [SDA99]. AWRI has focused on the problem of sensing location and as a by-product have produced the Location Service infrastructure which seeks to provide a robust and accurate location service, performs sensor fusion transparently for the user, and supplies reusable and extensible techniques to application programmers so that they may utilize location information in application-relevant ways [ABO02].

5.6.3 Counter Intelligence, MIT Media Lab

MIT media Lab’s Consortia on Things That Think (TTT) and their special-interest group on Counter Intelligence are primarily focused on single applications such as Smart Architectural Surfaces, Public Anemone, and an intelligent spoon, but have produced a distributed agents platform called Hive [ML03]. Hive is a Java-based framework that provides ad-hoc agent interactions, mobility, ontologies, and a graphical user interface to the entire distributed system. Hive is composed of three elements: *cells* which exist on each machine in the distributed system and provide the infrastructure connectivity, *shadows* which are the local resource encapsulation mechanism in each cell, and *agents* which utilize local resources through shadows and exist in one cell at a time. Hive has been used to develop a smart kitchen, a jukebox, and other applications and utilizes Java RMI for communication [MGR⁺99].

5.6.4 IBM Pervasive Computing Lab, IBM Research

The Pervasive Computing Lab under the direction of Bill Bodin at IBM Research in Austin, Texas, is performing work involving speculative integration to create *proof of concept* demonstrations utilizing modern technology to create such things as an advanced media living room, a networked kitchen, and integrated automobiles. Although not developing new technologies, they are combining existing technologies in new and interesting ways to show what is possible. For the designer and integrator it should be of interest that they use IBM Websphere, Java servlet technology, and Lotus Notes [Yor03].

5.6.5 i-LAND, AMBIENTE

The AMBIENTE division of the Fraunhofer-IPSI Research Institute in Germany is working on i-LAND, an Interactive LANDscape for creativity and innovation, to create cooperative buildings out of *roomware* components such as an interactive electronic wall, interactive table, and computer-enhanced chairs in order to create the offices of the future [AMB03]. The research focus is on these *roomware* components, but they are connected through integration on the COAST [SKSH96] cooperative hypermedia framework. COAST is groupware and heralds from research done in the Computer Supported Cooperative Work (CSCW) field. COAST is an object-oriented (SmallTalk) toolkit that supports the creation of synchronous groupware.

5.6.6 Interactive Workspaces, Stanford University

At Stanford University, the Interactive Workspaces Project is exploring work collaboration technologies in technology-rich environments with a focus on task-oriented work such as design reviews or brainstorming sessions. Their experimental facility is called “iRoom” where they are investigating integration issues with multiple-device, multiple user applications, interaction technologies, deployment software, and component

integration [SIL03]. To answer the integration and interactive workspace building challenge they have developed iROS, a middleware system for interactive workspaces. iROS is comprised of three subsystems: the *EventHeap* which provides coordination, the *DataHeap* which provides data movement and transformation, and *ICrafter* which provides user resource control. Through iROS they seek to provide a middleware layer that provides true platform portability, application portability and extensibility, robustness, and simplicity [PJKF03].

5.6.7 MavHome, The University of Texas at Arlington

The MavHome [You03] (Managing an Adaptive Versatile Home) project at the University of Texas at Arlington is focused on research involving the home as an intelligent agent that seeks to maximize the comfort of its inhabitants while minimizing resource consumption (e.g., power, water, natural gas) and maintaining safety and data security. MavHome environments currently include the MavLab where they conduct research daily, MavKitchen for kitchen environment experiments, and the MavPad, an on-campus apartment where they perform human-in-the-loop experiments. MavHome is comprised of many independent components written in C++ and Java integrated with CORBA mixing in ZEROCONF technology to replace the naming service and allowing self-publication and discovery services. The MavHome approach leverages existing technologies so they can concentrate on the main artificial intelligence focus of their project.

5.6.8 Microsoft Easy Living, Microsoft Research

The Vision Group at Microsoft Research is in the process of developing a prototype architecture and associated technologies for intelligent environments in their Easy Living project. Their research is concerned with using computer vision for inhabitant tracking and visual gesture recognition, sensor fusion, context-aware computing using geometric models, automatic sensor calibration, dynamic and adaptable user interfaces, generalized communication and data protocols, and system extensibility [MRVG03]. To provide integration for their research systems the Easy Living project is using middleware called InConcert that is being developed at Microsoft Research in conjunction with other groups there. InConcert was created out of a need for middleware that can provide communication in a distributed environment, but they feel it is better suited for the real-time and unusual needs of an intelligent environment than DCOM, Java, or CORBA. InConcert provides an asynchronous message-passing system with machine independent addressing and XML-based message protocols. Unique in this system is that the naming service is integrated in the message delivery [BMK⁺00].

5.6.9 Sentient Computing, Cambridge University

The Laboratory for Communication Engineering (LCE) at Cambridge University (originally in conjunction with AT&T Laboratories Cambridge, now shutdown) is pursuing their Sentient Computing Project with the focus of simulating computer perception in computing systems that detect, interpret, and respond to facets of a user's environment [Ber03]. Research has involved a deployed ultrasonic location system, advancements in world modeling including spatial considerations, and sentient computing applications such as world model browsing, remote desktop displays that follow users, smart posters using context-aware information retrieval, and ubiquitous user interfaces [LC01]. The omniORB CORBA package and VNC (Virtual Network Computing) both originated from the AT&T Labs, Cambridge, research.

The Sentient Computing group no longer uses omniORB in favor of middleware that can support context-aware multimedia applications called QoS DREAM, also under research and development at the LCE. QoS DREAM supports multiple types of sensors and provides a simple spatial model for representing *locatable entities*, real-time model and sensor data integration, an event mechanism for notifying applications of location information, a query-able location database, and an ease of extensibility. Like many others, this project is also developed in Java and with Java technologies. QoS DREAM is based on providing Quality of Service guarantees and takes a location-centric approach to the services it provides [NCM01].

5.6.10 Smart Space Lab, NIST

In the United States, the National Institute of Standards and Technology (NIST) has setup the Smart Space Laboratory to “address the measurement, standards, and interoperability challenges” [NIS03] for *smart spaces*. This group has established a two phase approach to addressing their interests in pervasive computing. In the first phase they are identifying areas for standardization, creating real experiences in which to identify applicable measurements, and identifying security measures to ensure the privacy and integrity of systems. The second phase involves developing specific metrics, tests, and comparative data sets for the community, providing reference implementations of designed system ideas, collaborating with industry to form standard specifications, establishing industry and academic partnerships, and integrating the phase 1 technologies to explore distributed issues with the technologies. From these goals they are working to create a test-bed containing a defined middleware component that provides real-time data communications, a connection broker for sensors, and containers for processing data. NIST has already released their Smart Flow System that provides a data flow server, graphical interface, and control console as well as audio, video, and voice recognition interfaces [NIS03].

5.7 CONCLUSIONS

Perfection in software systems is a pursuit of many system architects and developers. Whether it can be obtained is an open question. So much really comes down to what the project needs, wants, and desires require. This is why requirement analysis is such a key part to software engineering. Simple and elegant designs seem to provide better solutions than complex ones. Middleware can certainly alleviate complexity in designs, development, and maintenance, but there is no silver bullet [Bro95].

Middleware should complement a project, making it easier to design, develop, and maintain. It should provide a level of interoperability, reliability, efficiency, throughput, security, scalability, adaptability, dynamism, availability, fault tolerance, flexibility, portability, maintainability, and reusability. Architects should know what the limitations and requirements are and choose middleware that best fits the goals of the application while watching out for proprietary and non-standard solutions that may leave the project trapped in certain technologies.

In this chapter we discussed the concept of middleware, presented the desirable characteristics, and discussed its forms as well as the positives and negatives of each form. We introduced some of the technologies in current use, developed an understanding of the importance of standards and where to find them, and presented design issues to consider to assist in choosing the proper form and technology for a project. We hope we have developed an understanding of the benefits of middleware. We ended by presenting what current intelligent environments use for middleware that may provide some insight, ideas, and options for other smart environment projects.

In conclusion, middleware solutions can provide a solid platform on which to develop smart environment systems given a thorough knowledge of requirements and middleware solution options.

References

- ABO02. Gregory D. Abowd, Agathe Battestini, and Thomas O'Connell. The Location Service: A framework for handling multiple location sensing technologies, 2002. Website: www.cc.gatech.edu/fce/ahri/publications/location_service.pdf.
- AEGO96. G. Abowd, J. Engelsma, L. Guadagno, and O. Okon. Architectural Analysis of Object Request Brokers. *Object Magazine, special issue on distributed systems*, 1996.
- AHR03. AHRI. AHRI - Aware Home Research Initiative, Oct 2003. Website: www.cc.gatech.edu/fce/ahri/index.html.
- AMB03. AMBIENTE. AMBIENTE Activity: i-LAND, Oct 2003. Website: www.darmstadt.gmd.de/ambiente/i-land.html.
- ANSI03. American National Standards Institute. American National Standards Institute, Oct 2003. Website: www.ansi.org.
- Ber96. Philip A. Bernstein. Middleware: a model for distributed system services. *Communications of the ACM*, 39(2):86–98, 1996.
- Ber03. Alastair Beresford. CUED: Laboratory for Communication Engineering, Oct 2003. Website: www-lce.eng.cam.ac.uk/research/?view=2&id=7.
- Blo92. John Bloomer. *Power Programming with RPC*. O'Reilly & Associates, Inc., Sebastopol, CA, February 1992.
- BMK⁺00. Barry Brumitt, Brian Meyers, John Krumm, Amanda Kern, and Steven A. Shafer. EasyLiving: Technologies for Intelligent Environments. In *Proceedings of the Second International Symposium for Handheld and Ubiquitous Computing HUC*, pages 12–29. Springer, 2000.
- Bro95. Fred Brooks. *The Mythical Man-Month*. Addison-Wesley, 1995.
- Che03. Stuart Cheshire. Zero Configuration Networking (Zeroconf), Oct 2003. Website: www.zeroconf.org.
- Coe98. Michael H. Coen. Design Principles for Intelligent Environments. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 547–554, 1998. Website: cite-seer.nj.nec.com/coen98design.html.
- CPW⁺99. Michael Coen, Brenton Phillips, Nimrod Warshawsky, Luke Weisman, Stephen Peters, and Peter Finin. Meeting the Computational Needs of Intelligent Environments: The Metaglu System. In *Proceedings of MANSE'99*, Dublin, Ireland, 1999. Website: cite-seer.nj.nec.com/coen99meeting.html.
- Ede94. Herb Edelstein. Unraveling Client/Server Architecture. *DBMS*, 34(7), 1994.
- Gri03. Duncan Grisby. omniORB, Oct 2003. Website: omniorb.sourceforge.net.
- Gro03. AIRE Group. MIT Project AIRE – About Us, Oct 2003. Website: www.ai.mit.edu/projects/aire.

- HV99. Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, 1999.
- IANA03. IANA. IANA Home Page, Oct 2003. Website: www.iana.org.
- IETF03. IETF. IETF RFC Page, Oct 2003. Website: www.ietf.org/rfc.html.
- Ins03. Software Engineering Institute. Software Engineering Institute (SEI) Home Page, Sep 2003. Website: www.sei.cmu.edu/sei-home.html.
- IoEEE90. Institute of Electrical and Electronic Engineers. IEEE standard computer dictionary: A compilation of IEEE standard computer glossaries, 1990.
- IOfS03. International Organization for Standardization. ISO - International Organization for Standardization, Oct 2003. Website: www.iso.ch/iso/en/ISOOnline.openerpage.
- ISAG03. CMU Intelligent Software Agents Group. Intelligent Software Agents, Oct 2003. Website: www-2.cs.cmu.edu/softagents/intro.htm.
- ITU03. International Telecommunications Union. The ITU Telecommunications Standardization Sector (itu-t), Oct 2003. Website: www.itu.int/ITU-T.
- LC01. AT&T Laboratories Cambridge. Sentient Computing Project Home Page, 2001. Website: www.uk.research.att.com/spirit.
- Lin97. David S. Linthicum. Next Generation Middleware, Jul 1997. Website: www.dbmsmag.com/9709d14.html.
- LS79. B. W. Lampson and H. E. Sturgis. Crash recovery in a distributed data storage system. Technical report, XEROX Palo Alto Research Center, 1979. Website: cite-seer.nj.nec.com/lampson79crash.html.
- MGR⁺99. Nelson Minar, Matthew Gray, Oliver Roup, Raffi Krikorian, and Pattie Maes. Hive: Distributed Agents for Networking Things. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.
- ML03. MIT Media Lab. MIT Media Lab: Projects List Database, Oct 2003. Website: www.media.mit.edu/research/index.html.
- MRVG03. Microsoft Research Vision Group. Easy Living, Oct 2003. Website: research.microsoft.com/easyliving.
- NCM01. Hani Naguib, George Coulouris, and Scott Mitchell. Middleware Support for Context-Aware Multimedia Applications. In *Proceedings of the Third International Working Conference on Distributed Applications and Interoperable Systems DAIS*, pages 9–22, 2001.
- NIS03. NIST. Welcome to the NIST Smart Space Laboratory Web Site, Oct 2003. Website: www.nist.gov/smartspace.
- OAS03a. OASIS. OASIS, Oct 2003. Website: www.oasis-open.org.
- OAS03b. OASIS. UDDI.org, Oct 2003. Website: www.uddi.org.
- OMA03. Open Mobile Alliance. Open Mobile Alliance, Oct 2003. Website: www.openmobilealliance.org.
- OMG03. Object Management Group. Object Management Group, Oct 2003. Website: www.omg.org.
- PJKF03. Shankar R. Ponnekanti, Brad Johanson, Emre Kiciman, and Armando Fox. Portability, Extensibility and Robustness in iROS. In *Proceedings of IEEE International Conference on Pervasive Computing and Communications*, Mar 2003.

- PLQ⁺03. Stephen Peters, Gary Look, Kevin Quigley, Howard Shrobe, and Krzysztof Gajos. Hyperglue: Designing High-Level Agent Communication For Distributed Applications, 2003. Website: cite-seer.nj.nec.com/peters03hyperglue.html.
- Rao95. B. R. Rao. Making the Most of Middleware. *Data Communications International*, 24(12):89–96, Sep 1995.
- RN95. Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 1995.
- Sch95. George Schussel. Client/Server Past, Present, and Future, 1995. Website: news.dci.com/geos/dbsejava.htm.
- SDA99. Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The Context Toolkit: Aiding the Development of Context-Enabled Applications. In *Proceedings of ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*, pages 434–441, 1999.
- SDP⁺96. K. Sycara, K. Decker, A. Pannu, M. Williamson, and D. Zeng. Distributed Intelligent Agents. *IEEE Expert*, 11(6):36–46, December 1996.
- SIL03. Stanford Interactivity Lab. Interactive Workspaces, Oct 2003. Website: iwork.stanford.edu.
- SKSH96. Christian Schuckmann, Lutz Kirchner, Jan Schummer, and Jorg M. Haake. Designing Object-Oriented Synchronous Groupware with COAST. In *Computer Supported Cooperative Work*, pages 30–38, 1996.
- SM03. Inc. Sun Microsystems. The Source Java Technology, Oct 2003. Website: java.sun.com.
- Ste95. Steve Steinke. Middleware Meets the Network. *LAN: The Network Solutions Magazine*, 10(13):56, 1995.
- STK02. James Snell, Doug Tidwell, and Pavel Kulchenko. *Programming Web Services with SOAP*. O’Reilly & Associates, Inc., Sebastopol, CA, 2002.
- TA02. Khai N. Truong and Gregory D. Abowd. INCA: Architectural Support for Building Automated Capture and Access Applications, 2002. Website: www.cc.gatech.edu/fce/ahri/publications/inca.final.pdf.
- TOG03. The Open Group. DCE Portal, Oct 2003. Website: www.opengroup.org/dce.
- UPn03. UPnP Forum. Welcome to the UPnP Forum!, Oct 2003. Website: www.upnp.org.
- Web03. Webopedia. SQL, Oct 2003. Website: www.webopedia.com/TERM/S/SQL.html.
- WWWC03. World Wide Web Consortium. World Wide Web Consortium, Oct 2003. Website: www.w3c.org.
- Yor03. Candice A. York. IBM’s Advanced PvC Technology Laboratory, Oct 2003.
- You03. G. Michael Youngblood. MavHome: Managing an Adaptive Versatile Home, Oct 2003. Website: mavhome.uta.edu.