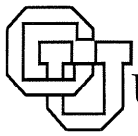
Software for Small, Communicating Computers

Gary J. Nutt

CU-CS-835-97



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Software for Small, Communicating Computers

Gary J. Nutt
Department of Computer Science, CB 430
University of Colorado
Boulder, CO 80309-0430
nutt@cs.colorado.edu

CU-CS-835-97 March 1997

University of Colorado at Boulder

Technical Report CU-CS-835-97
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Software for Small, Communicating Computers

Gary J. Nutt
Department of Computer Science, CB 430
University of Colorado
Boulder, CO 80309-0430
nutt@cs.colorado.edu

March 1997

Abstract

Computers have become an important part of today's everyday life, from their high profile use to manage corporate information, to personal information retrieval (web browsing), to information kiosks, to consumer electronics, and to managing our microwave oven. The spectrum of computers runs from the very large supercomputers for intensive computation to the very small to replace traditional electronic circuits. Today, there is exciting new computer science being applied to small computers. This tutorial addresses the evolving software technologies for small computers, specifically, those used in a distributed environment. These machines support multitasking continuous media environments, though their application domains are specialized for explicit purposes.

1 Introduction

Today, computers range from process control computers that are an economical replacement for hardwired logic to supercomputers designed to perform massive computations in the shortest possible time. The smallest computers have become pervasive in diverse applications such as controlling power supplies, games, automobiles, appliances, and device controllers for a computer. It now appears as though a more sophisticated level of *communicating* computers is about to become pervasive in society. This tutorial paper describes the rationale for developing these *small, communicating computers* (or SCCs) then considers the software that will control them.

1.1 The Evolution of Small, Communicating Computers

SCCs are at the confluence of two mainstreams of computer usage: distributed computing and embedded systems (see Figure 1). Distributed computing has become a mainstream computer science activity since 1980 (with the commercialization of the Ethernet). Since 1985 there have been huge computer science research programs driven by the need to study various facets of distributed computing to support scientific computation (including the DARPA Star Wars program, NSF High Performance Computing and Communication, and National Research Council Grand Challenge problems). Since 1990, corporations have radically changed their information systems (IS) and information technology (IT), evolving from purely centralized computing to distributed computing. A more recent emphasis on computer and network technology to support human collaboration (shared information, electronic meeting rooms, shared virtual environments) has also contributed to the interest in effective distributed systems. The competitive business environment — commerce — has also stimulated distributed computing through the rapid growth of information dissemination through entertainment media and the Internet. Mainstream computer science research is highly focused on all aspects of distributed computing, ranging across hardware, operating systems, databases, networks, programming languages, programming paradigms, and numerical methods.

Somewhat independently there has been an evolution in the way electronic control systems are designed and implemented based on the use of computers. Electronic circuitry has been a classic element of larger systems since the 1950s. As this area developed, it became clear that control systems could be built with discrete, digital logic (rather than the continuous analog logic used in early systems). Computers became important in this evolution when designers realized that they were

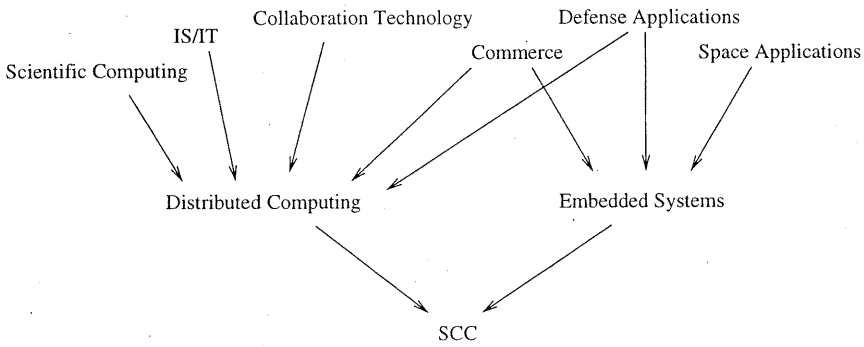


Figure 1: The Evolution of SCCs

a cost-effective way for implementing functionality that would otherwise be implemented in digital circuitry. By incorporating a microcomputer (such as an Intel 4004) into the controller design, then storing the software to control the computer in a ROM, the computer could perform the same function as hardwired logic. This avenue has been driven by commercial ventures in sophisticated control devices to "automatically" perform a broad range of functions ranging from microwave oven control to set-top boxes for television control. The defense and space programs have stimulated tremendous growth in these embedded systems through their intensive refinement of navigation and control devices from hardwired controllers to computers with sophisticated recognition and strategy selection mechanisms.

As the next century approaches, we are seeing striking interest in the application of small computers within a distributed computing environment. Part of the interest in SCCs stems from a trend of "thinning down" computer systems in general [15], and distributed systems in particular; embedded systems developers are interested in SCC technology since they have "fattened up" to the point that they need substantially more sophisticated systems to manage their software. There is also a new overlap of applications between distributed systems and embedded systems — that of supporting continuous media.

The result is a confluence of technologies; thinned-down distributed systems required the same computer technology as fattened-up embedded systems. The alignment of technologies coupled with the emergence of several markets has stimulated a very high level of activity in small, communicating computers.

1.2 Driving Applications

The rapid evolution of SCCs is driven by user requirements rather than technological innovation. SCCs have evolved because people want to buy them for their features (or perceived features) rather than because they are an application of a new technology. The following application areas are significant driving forces:

Ubiquitous Computing Weiser introduced *ubiquitous computers* in 1993 [14, 13]. The key elements of ubiquitous computers are that they are pervasive, nonintrusive devices that provide useful service while being essentially invisible to users. Further, they are fully connected into a global network. An important aspect of ubiquitous computing is that they are an enabler of collaboration from the perspective of situated work [14].

Interactive Television Interactive television (ITV) is expected to provide consumer services including basic TV, interactive entertainment, digital audio, video-on-demand, home shopping, financial transactions, games, digital multimedia libraries, and electronic versions of traditional print media [4]. The *set-top box* is the SCC in this application world.

Web Browsing The availability of information over the Internet grew tremendously in popularity beginning in 1996. Today, commercial advertisers, print media, sports teams, etc. frequently list a home page where detailed information can be retrieved over the Internet. Web browsers are emerging as a "hit application" driving a complete line of development. In today's technology, Netscape and Microsoft (among others) provide free copies of PC software to implement a *web browser*. A PC (or Macintosh) owner can load a web browser, connect the machine to an Internet provider over ordinary telephone lines, and then use the web browser as a user

interface for information retrieval across the Internet. In 1996 Web TV^{TM} began offer a product resembling a set-top box with a built-in web browser [12]. WebTV's display is the TV set, the input device is a remote control (or optional keyboard), and the 2-way communication network is the telephone network. It is a SCC especially designed to run only a specific web browser program.

Thin Client Machines There is another product trend intended to exploit the success of web browsers. The premise is that web browsers have established a human-computer interface (HCI) that is widely accepted in the consumer marketplace; therefore, the HCI could be used for services other than web browsing (including, for example, ITV). In general, this leads to a family of SCCs that span the functionality between a dedicated web browser and a general-purpose computer — the exact trend we used to explain the emergence of SCCs from distributed systems. The terms *thin client* and *network computer* are used to describe these machines — ones that depend on HCI technology similar to a web browser, but where additional software is incorporated into the machine. The Sun JavaStation product line is intended to operate in this environment [1].

Evolving Embedded Systems These applications have evolved due to the need to use computers to automate increasingly complex tasks in a hostile environment. For example, the SCC may be an in-flight computer for spacecraft or a weapon or control a robot working in a mine. The work performed by these SCCs is well-characterized by navigation and control applications. The emerging work on telepresence over the network and agent-based automation takes advantage of the approach [10]. In these applications, the embedded system reads onboard sensors regarding attitude, direction, etc., and assists in flying the craft within acceptable value ranges. It could also respond to external commands to follow a particular course, etc.

In the remainder of this paper, we consider the software for SCCs. In Section 3 we focus on the application programming level, and in Section 4 we consider the operating system technology.

2 SCC Application Domains

SCCs are evolving from distributed computing and embedded systems. While there is significant commonality, there are traditionally differences in the environments of the two families of systems.

2.1 Embedded Systems Extensions

Embedded systems exist as a component in some larger system such as a spacecraft, weapon, or microwave oven (see Figure 2). From the total system point-of-view, the embedded computer (including its software) is just another component in the set of components that make up the total system. An important characteristic of most of these components is that they interact with one another to implement a function for the total system. Thus, a sensor might be one component and an activator another; when a sensor in Component B detects some stimulus, the embedded computer is required to produce some response to the stimulus within a predefined period of time — the embedded computer must operate under a *hard real-time constraint.* In the most critical of these cases, failure to react to a stimulus within the real-time constraint can result in catastrophic

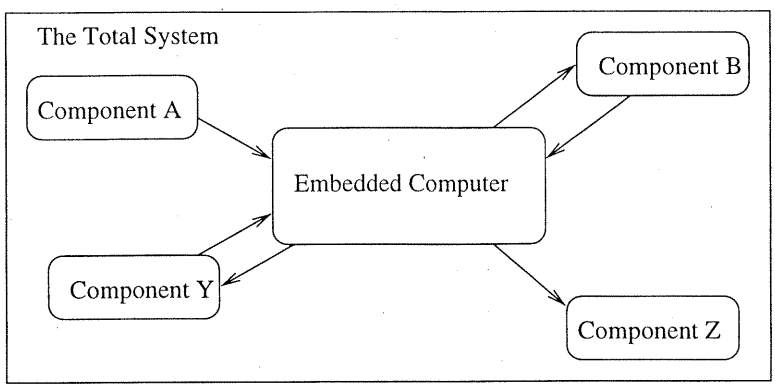


Figure 2: Embedded System Organization

failure. It is easy to imagine these kinds of failure in navigation and control applications in an onboard computer for a spacecraft.

Other functions performed by the embedded computer might have real-time constraints, but if the computer fails to meet the scheduling constraint, the failure is undesirable but not fatal. In a PBX system, the digital switch embedded computer is expected to route incoming voice packets from one line to another within a real-time constrained deadline. However, if the digital switch occasionally drops a voice packet, the digital-analog converter component can be designed to interpolate the values in the missing packet and still deliver acceptable-quality audio information. In this case, we say that the embedded computer operations under one form of a *soft real-time constraint.*

Because the computer is embedded in a hardware environment, the load provided by the other components typically has a unique property: it can be characterized as a set of periodically recurring tasks. The components in the system periodically take measurements, then require the computer to analyze the measurements to provide a desired reaction. Much of the real-time technology used in traditional embedded systems is based on this *periodic work*, whereas SCCs will have periodic, *sporadic*, and sometimes *aperiodic* workloads.

The environment for an embedded computer system is well-defined by the total system in which it exists. Because the embedded system traditionally deals with other hardware components — sensors, actuators, controllers, etc. — it must be designed to handle hard real-time constraints. As the nature of the total system applications change, there is an increasing need to handle soft real-time for processing audio and video streams of information. This need to support *continuous media* is a major influence in causing the overlap of distributed computing and embedded systems.

2.2 Client-Server Distributed Systems

Client-server computing is the dominant model for how application software can be distributed across two or more general-purpose computers (see Figure 3). This logical model was derived from early experience on the ARPAnet in the 1970s to accomplish file transfer and remote terminal access. In the client-server model, the overall computation is partitioned into two parts, one part to be executed on a *client* machine and another part to be executed on a *server* machine. The client represents an *active* entity and the server a *passive* entity; the server can do work when it is asked, and the client decides when to invoke the server.

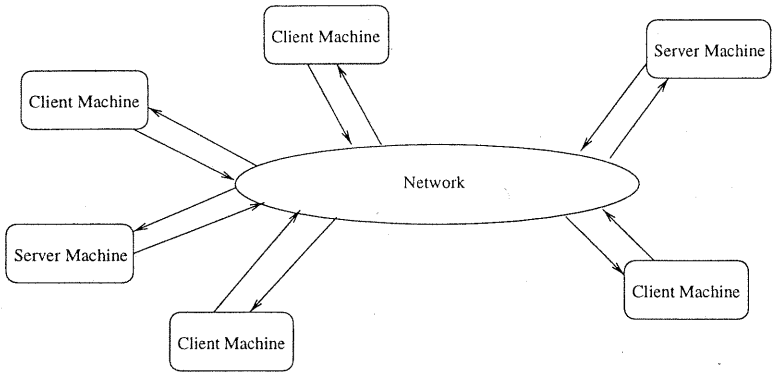


Figure 3: Distributed System Organization

Experience with the basic model has encouraged designers to create the system so that the server supports many clients, and hence, to implement functions that manage shared resources. The earliest commercial servers were the disk and file servers to manage shared information (see Chapter 16 of [9]), print servers to manage a shared printer, communications servers, to manage a shared data multiplexor connected to modems, etc. Another widely-used application was to create a "compute server" in a network of inexpensive client machines; each client implemented a window system and other elements of an HCI, while the compute server provided cycles to execute actual applications. For example, the X window system is built on this model.

The client-server model was also a natural way to accommodate the evolution from "legacy mainframe computing" to distributed models (see [8]). It provided a technological evolution path that allowed organizations to begin displacing their large mainframe computers with a network of clients and servers, where the servers were initially the extant mainframe computers. This approach is particularly valuable to business information systems since it allowed them to adopt a strategy for evolving their timesharing and transaction systems into a scalable distributed computing environment.

The liabilities of client-server computing are in its complexity. In general it has become a challenge to design good ways for software to be distributed between a client and a server. Once the software is partitioned, its performance may decline dramatically. Client-server networks introduce a new level of complexity in administering the computer system. An effective distributed environment in which the clients and servers intercommunicate using well-accepted network protocols such as remote procedure call require considerable attention to set-up and maintain. This administrative complexity is especially problematic in cases where the client machine is to be used in nontechnical environments, e.g., the home. The trend toward thin clients is intended to simplify these complexities, especially in administering the client machine.

As the client-server model has evolved, the implicit goal has been to offload increasing amounts of the computation from the server (mainframe) to the clients (PCs). This allows the overall network to scale without depending on the scalability of the server — the shared resource. The recent trend to thin down clients is a radical departure from the traditional goal of distributed computing, but it has an ever-increasing list of proponents. This trend to of *selectively* assigning work to the client machine based on the way the user applies the machine provides the momentum for thinned-down clients and SCCs.

Finally, note that a significant difference between embedded systems and client-server computing is in the nature of the offered load. While embedded systems can usually be designed around periodic work, client-server environments typically interact using a sporadic or aperiodic work patterns.

2.3 SCC Domains

Small, communicating computers are evolving to address the needs of thinned-down clients and fattened-up embedded systems. In turn, developers from these two camps are changing their requirements for computing support based on the evolution. For example, embedded system developers are expanding their application development horizons to include capabilities often found on a PC in addition to their standard requirement for real-time computing. PC-oriented developers are now looking to real-time techniques to support continuous media.

The convergence of technologies invites new application domains that were not previously supported by computers; ITV is a notable example of this.

As an example of a typical SCC application, we briefly review one of our own experimental systems that relies on the SCC technology.

The FLOATERS blimp is an unoccupied air vehicle developed to explore several facets of SCC software technology [10, 11]. FLOATERS is physically about 4 feet long, 3 feet tall, and 2 feet wide. The onboard computer is a 80486 laptop connected to other microprocessors to control components of the blimp. There are two propeller motors that have individual speed controls, and a ganged pitch control. By controlling the individual speeds and the pitch, FLOATERS can be navigated in all three dimensions. The blimp also contains equipment to report the location and orientation of the blimp, and a videocamera to view the terrain immediately in front of FLOATERS. Finally, there is an RF LAN so that the onboard computer can communicate with a ground station.

FLOATERS is used to develop and test ideas regarding continuous media support (communicating the videocamera image to ground stations), digital control (all navigation is done through the onboard computer), and agent-based automation (the onboard computer can be given abstract commands which it must translate into low-level commands for the propellers).

The onboard computer is a SCC. It has many elements of classic embedded systems in terms of the navigation and control tasks. However, it is also a distributed computation when we consider how it is to perform a high level task such as traversing a space according to some plan determined on the ground.

3 Application Software

There can be several different types of software in a computer system based on the way the computer will be used (see Figure 4). Each application domain uses a particular set of *vertical application programs* to meet its specific requirements. For example, the ITV domain uses TV guide software, set control software, etc.; a navigation and control system uses software to plot a course and other software to control onboard instruments.

Middleware is a term that has been coined in the last five years to identify a common set of functions that applies to a set of applications from similar domains. For example, ITV and web browsers may have a common set of software functions to decode and encode network packets, whereas onboard and weapons computers might use an entirely different set of network protocols

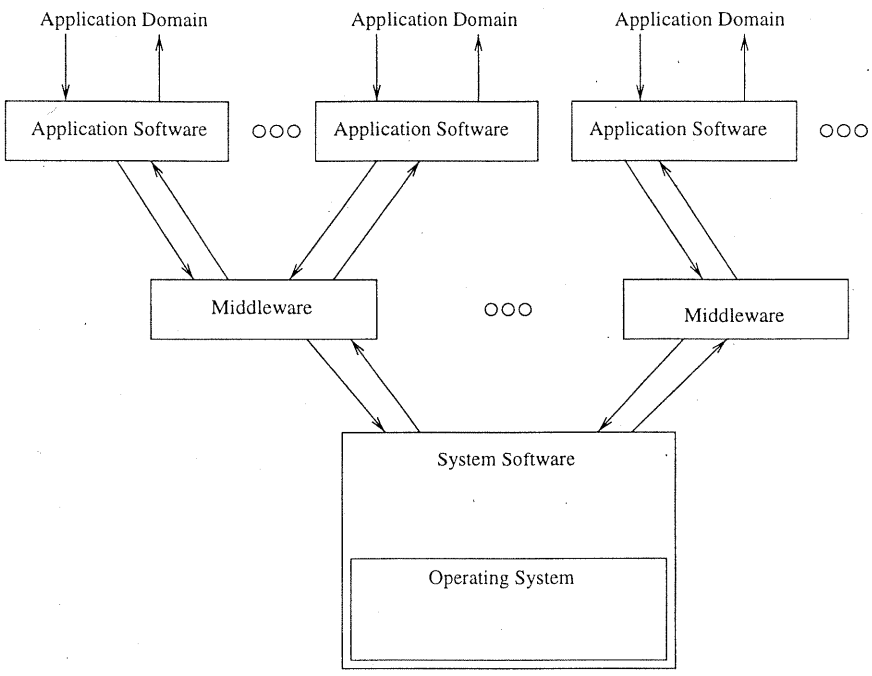


Figure 4: Software Classifications

(and hence have different middleware). Middleware has been popularized by taking advantage of the fact that some software can be reused across multiple applications and multiple domains. Web software is an example of middleware.

System software provides a common set of facilities for use by all middleware, and hence, by all applications in all domains. It includes various tools, window systems, file managers, along with an operating system. The operating system, discussed more in Section 4, is responsible for managing hardware resources and for providing software abstractions of the hardware to the middleware.

Application software is written to use an *application programming interface* (or API) created by the middleware and/or the system software. The nature of the API dictates much about the ultimate capability of the applications, and also about the style in which application software will be constructed. For example, if the middleware provides a particular menu system, then all applications will use that menu system for the HCI (meaning that they will all have a consistent "look and feel" with regard to menu operations).

3.1 Trends in Software for Embedded Systems

Embedded systems for controllers and onboard computers have been viable technologies for over 15 years. The challenge in these programming environments has been along two dimensions: Making the code fit into a limited amount of memory, and making the code execute in a limited amount of time.

Assembly Language Programming. When the amount of software in the embedded system was small, software for embedded computers was typically written in assembly language. This

allowed the programmer to be highly aware of the effect of the source code on the amount of memory being used by the program, and the amount of time that could be expected to execute the code. Unfortunately, this style of code development was very time-consuming, error-prone, and expensive to develop. The resulting code was also very difficult to maintain or modify to incorporate new functionality. However, success using this approach stimulated the idea of incorporating increasing amounts of functionality into the software. As the functionality requirements increased, the programming time increased at a much faster rate. Assembly language programming became impractical in the face of growing functionality requirements.

In mainstream computer science evolution, high level programming languages have completely displaced assembly languages. High level languages allow programmers to work at a much more abstract level than do assembly languages; with high level languages, programmers can devote more of their energy to designing innovative algorithms and solutions than is possible using assembly language. Before high level languages could dominate, it was necessary for the language translation (compiler) technology to become efficient enough that the space and performance losses due to the use of the abstraction were outweighed by the increased efficiency at the algorithm level (and in the time saved on programming itself[1]).

Single Threaded Software. The original software for an embedded system was written as a single program to be executed by the CPU in the embedded computer. That is, the requirements for the software could be identified, then a single program would be written to satisfy all the requirements. As requirement sets began to grow, the complexity of the control flow in the software became at least, if not more, complex than the requirements. For example, if code modules f_1, f_2, ..., f_n were designed to meet requirements r_1, r_2, ..., r_n, then a main program needed to be written to call f_i whenever appropriate. In the case that there were timing dependencies on the execution of the f_i, the situation could worsen to the point that any particularly function, f_i, might have to be decomposed into subfunctions $f_{i,1}$, $f_{i,2}$, ..., $f_{i,m}$, then to have $f_{i,j}$ called at just the right time. The main program is responsible for implementing this coordination; thus by its nature it is fragile, making it difficult to maintain or change.

Programmers soon realized that this could be handled much more effectively, i.e., greatly simplifying the construction and maintenance of the main program, by changing the *single thread of execution* into multiple concurrent threads of execution — *multithreaded execution.* Each of the f_i could be written as a separate program, being executed by a logical machine, using interrupts and synchronization events to start and stop the execution of the subfunctions, $f_{i,j}$. Then, a scheduling entity could simply run each $f_{i,j}$ when it was logically ready to run. This solution was also being widely used in the mainstream software technology in the 1970s, so it was a natural evolutionary change in embedded system software.

In a multithreaded environment, the programmer focuses only on implementing f_i as a set of subfunctions, $f_{i,1}$, $f_{i,2}$, ..., $f_{i,m}$ to be executed by a single thread in the multithreaded (or *multithreaded*) environment. Each thread is then assigned to one of the f_i.

Figure 5 represents the way the subfunctions are are executed on a machine having only one physical processor. Suppose that all of f_1, to f_n are ready to run; the operating system scheduler chooses one of them, say f_1, to use the processor. When subfunction $f_{1,1}$ has been completed, $f_{2,1}$

[1]Classically, programming language experts have argued that the increased ability to write *correct* programs justifies the use of high level languages

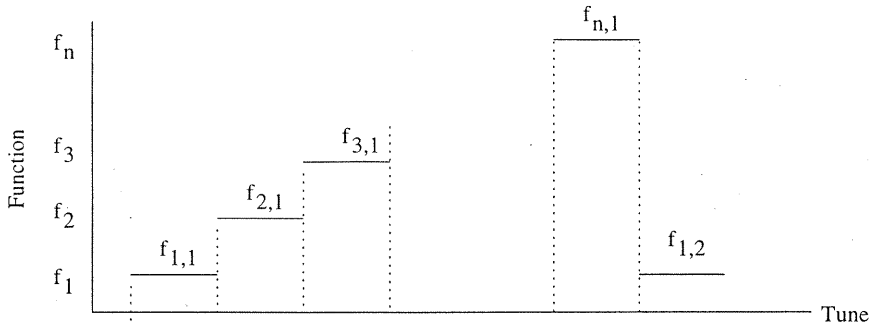


Figure 5: Executing Multiple Threads in a Single Processor System

begins execution, then $f_{3,1}$, etc. After $f_{n,1}$ has finished, (and presuming that the thread for $f_{1,2}$ is now ready to execute), the operating system runs the thread for $f_{1,2}$.

Single Address Space Computations. Multitasking/multithreaded environments quickly highlighted a new problem: the different threads implementing f_1, f_2, ..., f_n are all loaded into the machine's memory at one time, even though they implement independent functions. Therefore, when f_i was being executed, a subtle bug in the software might cause its thread to overwrite the part of memory being used by a different function, f_k. f_i might look like it is working correctly, but f_k would periodically fail for no apparent reason.

The mainstream software solution to this is for the hardware to provide memory protection mechanisms to prevent the thread executing f_i from reading or writing memory other than that which is allocated to it. This avoids the problem of f_i's thread overwriting the memory space used by f_k's thread since it is no longer possible for the thread executing f_i to even access the memory being used by the thread executing f_k (see Chapters 6 and 11 of [9]). Mainstream compiler technology evolved with the hardware to have translation systems construct f_i's program within its own *address space*.

Unfortunately, the address space isolation mechanism also prevents f_i from cooperatively passing information to f_k by writing information into a shared memory location. Such sharing is common in an embedded system whenever f_i and f_k are cooperative threads, working on a common problem (as is often the case in an embedded system). The usual solution is to completely bypass the memory protection mechanism by constructing f_i and f_k so they use the same address space, but execute as separate threads.

Neither approach is completely satisfactory. A better approach is to construct each thread so that it has its own address space, but so that it can selectively share parts of its address space with other threads (by mapping the shared part to shared memory). This is the contemporary mainline computer science solution to the problem, and it is also being used by more and more embedded software.

Time and Space Needs. Multithreaded/multi address space technology abstracts the memory space and execution time from the programmer. Experienced C programmers are still able to construct their code so that they can determine space requirements, but control on execution time is lost with the abstraction. (It was also true that the growing complexity made it essentially impossible to construct solutions that met timing constraints in assembly language.) This lead

embedded application programmers to begin using *real-time operating systems* to ensure that the various subfunctions are executed prior to some deadline established by the system requirements. From the programmer's point of view, this requires that the function specification identify the *frequency* a subfunction should run, the *time to execute* the subfunction, and a *deadline* by which the subfunction must be completed — hard real-time software.

Cross-Development Environments. Today, multithreaded, multiple-address space software for embedded systems is developed on a general-purpose computer, e.g. a separate UNIX, DOS, or Windows-based development computer. The compiler is a *cross-compiler* that runs on the development machine, but generates machine language programs for the target SCC. Embedded systems are developed almost exclusively using this code development model.

3.2 Thin Applications

> *What's the big deal about Java and the Web? The fact that they mark the death of fatware and the birth of dynamic computing built on rented components.* Edward Yourdon, [15]

In Section 1 we explained how SCCs have evolved from distributed computing by a recent trend toward loading less software onto the user's machine (contrasted with the trend where software packages are becoming increasingly large.) While one part of computing is to configure the computer with a faster processor, more RAM, and a larger disk; the trend in SCCs is to use limited memory, a network for accessing services, and to load software into the SCC only when it is needed. Hamilton [6] and others refer to this as a shift to *net-centric computing.*

Encapsulated application technology has emerged as a commercially viable way to produce applications for SCCs (as well as other classes of computers). The principle for this style of programming is that the hardware environment is a distributed environment made up of client and server machines. Server construction is accepted as being a software-intensive task, meaning that the construction of the software can be difficult, and the resource requirements to execute server code can be significant. Clients are lightweight entities that can cooperatively execute software by downloading an encapsulated application — called an *applet* — that has been especially designed to conduct the interaction between the client and the server, with a separate interaction between itself and the client environment.

Figure 6 pictorially represents the relationship among components in a systems supporting encapsulated applications. A client application and a server application intend to communicate to jointly perform some work. For example, the client application might be a user interface for browsing a database, and the server application might be the database. The applet has been written by the developers of the server application. The two pieces of software are designed to communicate with one another over the network. Next, the applet has also been designed to interact with the client application through a procedure-call interface (much simpler than the network interface between the server application and the applet). Now, when the user wants to use the server application, the server downloads the applet into the client application. When the user queries the server application, the client application passes the query to the applet, which then interacts with the server application to carry out the query.

The applet-based software environment is a key technology for allowing SCCs to be configured with modest resources yet be able to operate in a fully distributed computing environment. Of

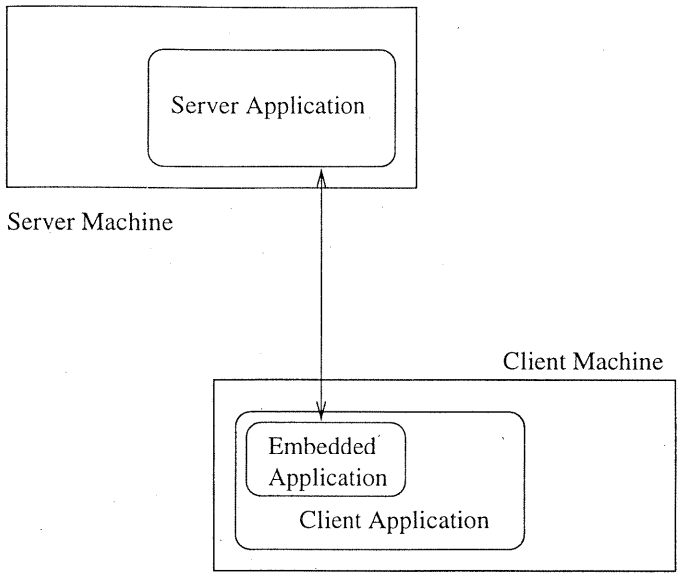


Figure 6: Encapsulated Applications

course it depends on there being a "standard" interface between the applet and the client application.

Java. Java programs are portable, object-oriented programs, described by Gosling (the language inventor) as "... C++ without guns and knives ..." [5]. This means that Java explicitly limits the ability to reference arbitrary objects within a program, a concession to help ensure secure operation of programs written in Java.

When Java programs are compiled, they are translated into a pseudo code language ("byte-codes") rather than into a native machine language. This means that compiled Java programs cannot be executed directly on target hardware, but that they must be interpreted by another package that has been implemented on the target hardware; this interpreter is called the *Java Virtual Machine* [7]. Any machine that contains the Java Virtual Machine can be given a copy of a compiled Java program, and it can then interpret the program.[2]

A Java Virtual Machine can be implemented in any environment, e.g., as an ordinary process in UNIX or as a part of a web browser. Web browsers such as Netscape Navigator support embedded applications by incorporating a Java Virtual Machine in the browser (see Figure 7). As a consequence, when the browser contacts a server to read an html file, the server can provide specialized functionality by downloading a copy of a Java applet — the compiled version — into the browser. The browser can then use the Java Virtual Machine to execute the program.

Web Browsers. The World Wide Web (WWW) was originally developed to support geographically dispersed set of collaborators [2]. The general idea is that there are a set of web servers that

[2]In cases where the speed of interpreting the applet is a limiting performance factor, the client can include a "just-in-time compiler" to translate the Java pseudo code into native language code; it can then be executed directly as if it were interpreted.

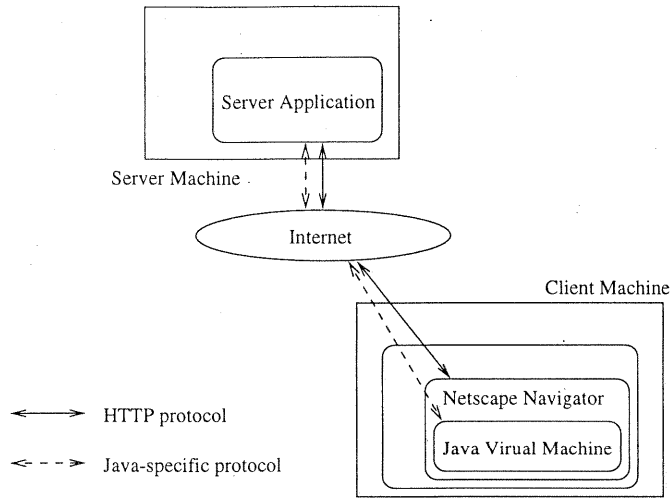


Figure 7: Running a Java Program as an Embedded Application

are information repositories, and another set of web clients that can be used to browse the information stored in the repositories. Information is referenced by having the client use a hypertext link to identify information within the web (i.e., by identifying a server and the information on that server). The WWW defines [2, 3]:

- An address system (Universal Resource Identifier, or URI) to reference information.
- The HTTP protocol for accepting requests and transferring information.
- The HTML markup language for describing information.

A web browser is a web client application that provides an HCI to allow an interactive user to issue a request to a server using a URI (or UR locators, URLs) using HTTP. The web browser will also use HTTP to accept the hypertext from the server. As the web has evolved, alternative network protocols including FTP, NNTP, Gopher, and WAIS can also been used, though the general behavior of the protocol is the same as for HTTP. When the web client obtains the HTML hypertext, it interprets it to create a new display for the interactive user. Just as there are alternative network protocols used in WWW the language used to describe the hypertext object may be be different from HTML. This new display may have URIs to other objects in the WWW space, allowing the user to issue a subsequent query, etc.

Notice that the web does not depend on any particular presentation of the HTML at the user interface; this means that two different web browsers will ordinarily display the same HTML information as two different displays. They should have the same "information content" but their own unique presentation.

It is also important to notice that WWW does not explicitly address encapsulated applications. HTML accommodates them by allowing the hypertext object to contain fields that are subject to arbitrary interpretation. This means that a server can place an applet program into a field of the hypertext, expecting the web browser to interpret it as, say, a Java or ActiveX applet. However, if the hypertext contains an applet, and if the web browser has the capability to execute the applet,

then the client and server will perform a new level of distributed computation over and above the basic capability assumed by the web.

4 Operating Systems

Part of the confluence of thin distributed systems and fat embedded systems is reflected in the evolution of operating systems (OSs) for SCCs. With a few exceptions (e.g., WebTV), SCCs are assembled using hardware from one source, an OS from a second source, and application software from many other sources (of course in the case of encapsulated software, applications are provided dynamically at run time, but the applet runtime environment is generally supplied by a different vendor that the ones supplying the hardware and OS). This section considers the OS technology.

4.1 OS Functionality

SCC OSs are required to be as compact and efficient as possible, yet are also required to provide a core multitasking, multi address space environment. Any OS can be systematically studied by considering the way it provides a general class of functions: process and resource management, device management, memory management, and file management [9].

Process and Resource Management. The process and resource management function is the heart of a multitasking OS. It create the software environment that has the illusion of multiple virtual machines to be used by the multiple tasks. Different OSs use different definitions of tasks, though the time-honored one is that of *process*. A process is an abstract entity that can have resources allocated to it, and that will execute a program. A *thread* is a similar abstraction, though threads are distinguished from processes in that they tend to use resources belonging to "someone else." In contemporary OSs, there is still the notion of a process that can have resources allocated to it, while a thread is a child of the process; the thread uses the process' resources. The OSs used in SCCs seem to take variants of this general approach to providing a multitasking environment.

Multithreaded operation is possible because the OS first defines a thread, then it provides a mechanism — called a *scheduler* — for time-multiplexing the processor across threads. When a thread is logically ready to execute (it has all the resources it needs at the moment, and is not blocked on any event), then it is said to be "ready to run." When the scheduler chooses to reallocate the processor, it selects from among the then-ready tasks, dispatching the task to the processor for a block of execution time. Later, the running task will be removed (because it was done or it was interrupted) and the scheduler will choose and dispatch another thread.

The resource management aspect of the OS is the part that decides which units of which resources should be allocated to a requesting entity, say a process. A process can only make a resource request while it is running (otherwise it is blocked and asleep). When the request is made, the resource manager decides if it can or cannot satisfy the request. If it can, it allocates the resource units and allows the process/thread to continue execution. If it cannot, the resource manager blocks the process/thread until some other process releases enough units of the resource to satisfy the request. At that time, the process can be made ready and begin competing for the processor again.

Device Support. Computers can be configured with a wide array of different device types. Some devices are low-speed character devices, e.g., a keyboard, touch-sensitive screen, or remote control; others are high-speed block devices, e.g., a packet network interface. The device manager function is to provide controlling functions for this array of devices.

The OS could be configured so that it has all possible device management strategies built into it; then when any device is added to the the system, the device management function (device driver) is already present. There are two criticisms of this approach: how does the original OS designer know what all the devices that anyone will ever want to add to the system? If obscure devices are not added to my system, why do I need to include their device drivers (using up precious memory space in my version of the OS)?

All contemporary OSs, including Microsoft DOS, use configurable device drivers. In the case of DOS, the OS is extended with various device drivers when the machine is booted. Most versions of UNIX allow device drivers to be added when the machine is powered down so they will be available the next time the machine is booted; recent versions provide a dynamically loadable device driver capability e.g., see Linux.

Embedded OSs are especially sensitive to this problem, and have tended to solve it by taking a variant of the UNIX approach (though, technically it is not quite as flexible). Some of these systems have an explicit configuration phase in which the OS is rebuilt to have the device management facilities for the specific hardware configuration.

Memory Management. The memory management system is responsible for allocating memory to processes, for protecting allocated memory from unauthorized access, and sometimes for automatically moving information back and forth between the memory and storage devices. Today's general purpose OSs use very elaborate *virtual memory* systems based on paging. These systems provide the most general functionality, accomplishing all the goals of allocation, protection, and automatic movement. However, there is a cost in time, space, and hardware for virtual memory. As mentioned in Section 3, embedded system designers frequently forego sophisticated memory management to conserve time or space.

In emerging SCCs, there will be fewer and fewer single address space application sets. Most will use use dynamic relocation hardware, but few will use full virtual memory in the immediate future.

File Management. Embedded systems typically do not use files, but thin clients make extensive use of them. This will be one of the parts of SCC OSs that will require innovation before a common OS can be used by the full spectrum of SCCs. Thin clients will continue to use files, and as embedded systems designers become acclimated to the use of the network (and as the network bandwidth increases), SCCs will increasingly make use of file access.

4.2 OS Organization

The software organization for how an OS is designed and implemented is relevant to the discussion of SCC OS technology. Prior to 1970, operating systems were typically constructed as one large monolithic block of code. Several different researchers began to advocate partitioning the functionality into modules that could be separately developed and maintained. This resulted in an approach in which core OS functionality was implemented in a *kernel*, with other aspects of the OS

being treated as ordinary user programs by the kernel. The kernel-organization is the main design style used in today's OSs (see Chapter 18 of [9]). UNIX is a common example of this approach.

As the need for OSs appeared in embedded systems, a new approach began to evolve: *configurable* OSs. In this approach, all OS functions are implemented in modules, based on a core module (similar in style to a kernel, but generally having less functionality than a kernel). As the embedded system software was designed, requisite modules were added to the core module. Finally, the core and required modules were combined and compiled into one specialized version of the OS — configured exactly to meet the requirements of the embedded system's application software. VxWorks is a common example of this approach; it is well-known among embedded system builders.

Today there is another technology that is being used to meet the general requirements for SCCs, including embedded systems: *microkernels*. A microkernel is a base module that has been compiled and exists an independent software entity at runtime; however, it is does not have enough capability to perform OS functions by itself. It is accompanied by a set of servers (same idea as servers in a client-server model, but these are microkernel servers). A microkernel is configured to a particular application programming environment by adding a set of microkernel servers to enhance the base functionality. In principle, these servers could be dynamically, added or deleted from the system, though that does not happen in practice. Microkernel OS technology is the state-of-the-art in OS organization. This approach is used in Mach, Chorus, and Windows NT.

4.3 Handling Time Constraints

The microkernel (or other atomic part of the OS) must implement the multitasking model. The scheduler is a key element of the model, since it determines how threads (or processes) will share the system's processor. Real-time requirements mean that the SCC OS must support a particular class of schedulers; if the scheduler cannot address processing deadlines in some manner, it cannot be used for embedded systems or continuous media environments.

Best Effort Scheduling. Web browsers and thin clients use *best effort* scheduling as is found in Windows 95 or UNIX. When a thread is ready to run, it is placed in a ready list; the scheduler selects ready threads for execution according to equitability concerns, i.e., over time, if N threads are ready, each will receive 1/N of the CPU time.

Hard Real-Time. SCCs require some form of real-time support, meaning that the scheduler must use priorities on each thread where the highest priority thread is guaranteed to run if it is ready.[3] Hard real-time schedulers can assure response by requisite deadlines by using a *rate monotonic* scheduler with periodic tasks. If there are other tasks that are sporadic, then hard deadlines can again be assured by using the worst case interarrival period that could occur in the set of sporadic tasks. Finally, if the tasks may become ready at random times — aperiodic tasks — then a sporadic server can be used with a rate monotonic scheduler to provide guaranteed service provided that the time between tasks arrivals in an aperiodic set is always greater than some minimum threshold. SCCs for embedded system applications require the OS to provide hard real-time scheduling support.

[3]The classic problem with such schedulers is that *priority inversions* can take place in which a high priority process is prevented from becoming ready while it waits for a signal from a ready lower priority process.

Soft Real-Time. SCCs for ITV, involving continuous media support such as MPEG stream decoding, also use deadlines for task completion. However, an ITV application will not necessarily fail if a deadline is missed. This allows the operating system to take a *soft real-time* approach, which is far less conservative that hard real-time; as a consequence, it makes much better use of the SCC's resources. Unfortunately, soft real-time is a leading edge and evolving technology, so the detailed requirements for the operating system are not known at this time.

5 Conclusion

The trend to thin down clients and to fatten up embedded systems is creating a new class of small, communicating computers with their unique software requirements. SCCs will be used in a spectrum of rapidly growing application niches, including ubiquitous computing, ITV, web browsing, thin client applications, and communicating embedded systems.

The technology business sector is rapidly repositioning itself to address this potentially huge market. This repositioning causes companies to reconsider the hardware designs, but more importantly, to completely rethink the way they write and support application software. This has strong implications on the nature of the operating systems used in SCCs, since they must be small and tailored to SCCs.

There are technological challenges to be overcome, but none appear to be impossible to address. The next 5 years will see an exciting growth in the SCC market, and a radical change in the way computer software is designed to address these machines.

References

[1] Javastations — an overview. WWW page at http://www.sun.com/961029/JES/whitepapers.orig/, 1996.

[2] Tim Berners-Lee, Rober Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. The world-wide web. *Communications of the ACM*, 37(8):76–82, August 1994.

[3] Michael Bieber and Fabio Vitali. Toward support for hypermedia on the world wide web. *IEEE Computer*, 30(1):62–70, January 1997.

[4] Borko Furht, Deven Kalra, Frederick L. Kitson, Arturo A. Rodriguez, and William E. Wall. Design issues for interactive television systems. *IEEE Computer*, 28(5):25–39, May 1995.

[5] James Gosling. Java language tutorials – your first jolt. Videotape, 1996.

[6] Marc A. Hamilton. Java and the shift to net-centric computing. *IEEE Computer*, 29(8):31–39, August 1996.

[7] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification.* Addison Wesley, 1997.

[8] Gary J. Nutt. *Open Systems.* Prentice Hall, 1992.

[9] Gary J. Nutt. *Operating Systems: A Modern Approach.* Addison Wesley, 1997.

[10] Sam Siewert. Operating systems support for parametric control of isochronous and sporadic execution streams in multiple time frames, 1996. Ph.D. dissertation proposal.

[11] Sam Siewert and Gary J. Nutt. A space system testbed for situated agent observability and interaction. In *The 5th Intl. Conf. and Exposition on Engineering, Construction, and Operations in Space and The 2nd Conf. and Exposition/Demonstration on Robotics for Challenging Environments*, June 1996.

[12] Webtv technical specifications. WWW page at http://www.webtv.net/HTML/home.specs.html, 1996.

[13] Mark Weiser. Hot topics: Ubiquitous computing. *IEEE Computer*, 26(10):71–72, October 1993.

[14] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, July 1993.

[15] Edward Yourdon. Java, the web, and software development. *IEEE Computer*, 29(8):25–30, August 1996.