

2 Distributed Systems

Distributed systems are the backbone of modern computer configurations, particularly configurations in which SCCs exist. In today's world, computers systematically rely on remote computers to store information, to perform specialized calculations, or to otherwise work together to implement an information technology solution. Tanenbaum describes these distributed system as

“A distributed system is a collection of independent computers that appear to the users of the system as a single computer.”¹

A distributed system is typically a collection of autonomous devices (computers) that are interconnected by some means for exchanging information. Systems people often think of a distributed system as being a collection of computing **nodes** interconnected by a **network**. Informally, you can always think of a node as being some kind of a computer, each with a communication device that connects it to a shared data switching network. Thus, collections of computers connected with a LAN or the Internet are the basis of distributed systems.

Tanenbaum emphasizes that the OS in a distributed system is designed so that when an application program is executed on the distributed system, it could use various nodes in the network without any special programming. That is, one important goal of a distributed system is that a program that runs on a single computer can also run (without modification) on a distributed system where it may use many computers during its execution. Programmers say that the distribution of the execution over multiple computers is *transparent*, meaning that the way that the program is written is the same whether it executes on a single computer or on the distributed system. In reality, contemporary distributed systems do a pretty good job of making the distribution transparent, but it is usually not perfect: the program may need minor modifications to execute on a distributed system – the amount of modification required reflects the closeness of the real system to a truly distributed system.

People are inspired to use distributed systems for a several primary reasons:

- It might be possible to execute a conventional program in less time on a distributed system than on an individual computer, since multiple computers could be working on a single problem at the same time.
- A distributed system makes it possible to incorporate specialized computers to perform specialize functions – possibly more efficiently or producing a better result. For example if an organization needs frequent access to a diversity of data, a distributed system could incorporate a computer that is dedicated to implementing a database. We say that such computers provide specialized *services* to other general purpose computers.
- It is sometimes beneficial to place a computer “close” to the source of its data. In physical closeness, this can enable the computer to validate and pre process data that it has collected prior to storing it for general use. For example, a point-of-sale terminal can manage on optical bar reader, enabling the terminal to detect and correct read errors without interacting with a computer that keeps track of total sales. Another interpretation of “close” has to do with logical proximity: organizations often prefer to store information for which they are responsible in a computer that they control. For example, a test lab might prefer to keep data related to product testing on a test lab machine rather than saving it in a centralized database or file system.
- Reliability can be built into a distributed system by duplicating computers to perform critical functions. For example, if a computer manages a corporation's transactions, the corporation may decide it is worth the cost and effort to duplicate each transaction to avoid information loss due to a machine failure.
- Distributed system designs can sometimes enable a system to scale its resources (such as processing capacity) to match its program execution load.

As a result of these (and other) reasons, and because of the evolution of technologies (such as networks), distributed systems are a cost-effective way of configuring computational facilities.

¹ [Tanenbaum, 1995], page 2.

2.1 Networks

Since distributed systems are collections of communicating computers, network technology is one of their essential cornerstones. [Section 1.3](#) introduced networks; in this section we will take a closer look at the technology to see how network devices are similar and different to/from other I/O devices, and to see how network technology plays such an important role in distributed systems.

2.1.1 Network Devices

Networks differ from many other devices in the way that the software views a network device. One could install a network device on a computer, but it would be useless unless the device was then connected to an actual data transmission network. Equally obvious, the network would still be worthless if there were no other computers attached to the data transmission network. A network device depends on the existence of an external, shared mechanism that can transmit and receive signals to/from designated recipients; we will refer to this data transmission mechanism as the **subcommunication network** (see Figure 2-1, which is the same as Figure 1-10).

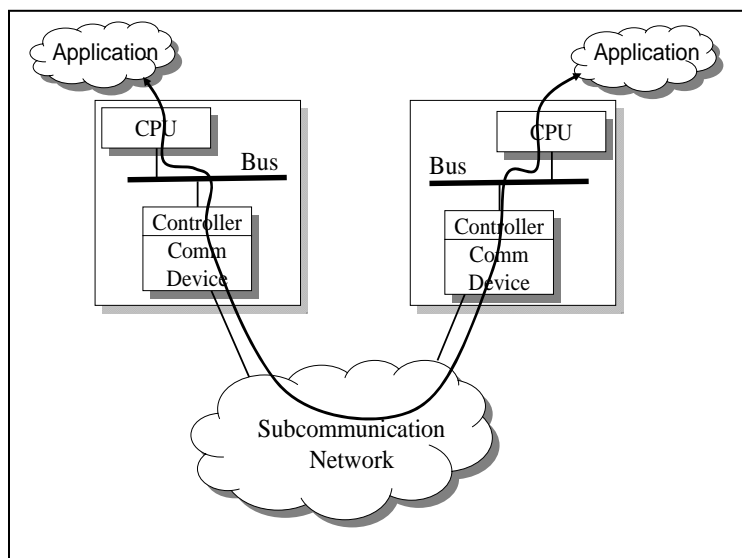


Figure 2-1: Network Communication

Notice that in the case of a network device, the actual *device* is the subcommunication network; that is, in the network case the actual device is not really part of the computer's configuration at all. The computer's component is the device controller that adapts the computer to the subcommunication network, often referred to as the **network interface controller**, (or **NIC**). The NIC's job is to be able to physically and electronically connect to the subcommunication network, and to be able to write data to (and read data from) the network. For send (or output) operations, the NIC translates data from the computer's internal binary form into the subcommunication network's internal form, and then transmits the data over the subcommunication network to a specific remote node. On input operations, the NIC receives data that is directed to it by the subcommunication network, and then translates it from the subcommunications network's internal form into the computer's internal form. NIC I/O commands are similar to commands for other devices, namely to read and write data from/to the subcommunication network. Part of what makes this tricky is that when one computer's NIC writes data, then the receiving computer's NIC must be ready to read the data, since the subcommunication network only transfer information, but does not buffer it until the recipient is ready to receive it.

This means that the NIC is normally designed to receive information that is transmitted to a remote computer even if the software on the receiving computer may be busy doing some other task. The NIC buffers incoming network data until the host computer's software issues a read operation on the NIC; only then does the NIC return information from its buffers (sent by a remote computer) as a result of the read. Obviously, every NIC has a fixed amount of memory that it can use for a buffer. Therefore it can only

store a fixed amount of information. It is important that the host computer be prepared to receive that information within a short amount of time – otherwise the NIC buffer memory could overflow and data will be lost. Generalizing this line of thought, you can see that two computers can only communicate if they agree on a spectrum of issues, ranging from *when* will data be transmitted from one computer to another, *which* computer will transmit and which will receive, *what* will be the format of the data, and so on.

Standards can be used to establish the nature of the behavior of communicating computers. Thus, standardization is very important in network communication. The International Standards Organization (ISO) Open Systems Interconnect (OSI) Architecture (introduced in Chapter 1) is a general *framework* in which more detailed standards can be specified to facilitate common communication.

Let's first focus on the NIC device: from the software perspective, the device controller is just like any other input and output device controller in that the software can write information to the controller and it can read information from the controller, even though there is really no "device" incorporated into the controller. The subtlety of the network arises from the amount of knowledge that can be incorporated in this "device" – Figure 2-2 illustrates the local host's device-oriented view of a person using a remote computer. The local computer's interface to the remote computer is the NIC, but the NIC's "device" is actually a subcommunication network that can be used to communicate with any of a number of remote computers, some of which are attended by a human. If the local host asked a question of the "NIC device" the "device" could answer with the full intelligence of the user at the remote computer. This is the sense that makes NIC devices have an unusual behavior – the amount of intelligence in the logical device depends on the remote host, not on any local physical device hardware.

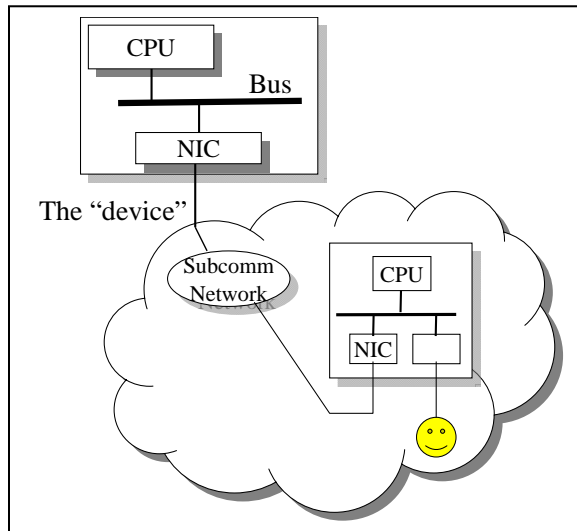


Figure 2-2: A Local Host's Perspective of a Remote Host and User

This dependence on the potentially complex behavior of the remote computer is the reason that the ISO OSI model is important. Ultimately, the OSI Architecture defines a layered set of behaviors that the local and remote hosts implicitly agree to use. Each level has an implied level of intelligence, for example, if the remote host computer really behaves like a digital camera, then the local host computer need not be prepared to discuss anything more abstract than what a digital camera could understand. But if the remote host computer is prepared to execute a remote procedure for the local host computer, then the nature of the interaction between the two computers must contain substantially more detailed information.

Figure 2-3 represents the OSI Architecture levels. In the OSI model, two computers can communicate with one another if they are designed to operate using the same protocol at the same OSI layer. For example, we might say that computers communicate at the data link layer using the Ethernet protocol if the two computers exchange information using an Ethernet LAN. As other examples, two computers might communicate at the transport layer using TCP; or they might communicate at the presentation layer using a

remote procedure call protocol. We will have much more to say about the details of particular layers and protocols as we learn more about programming distributed systems.

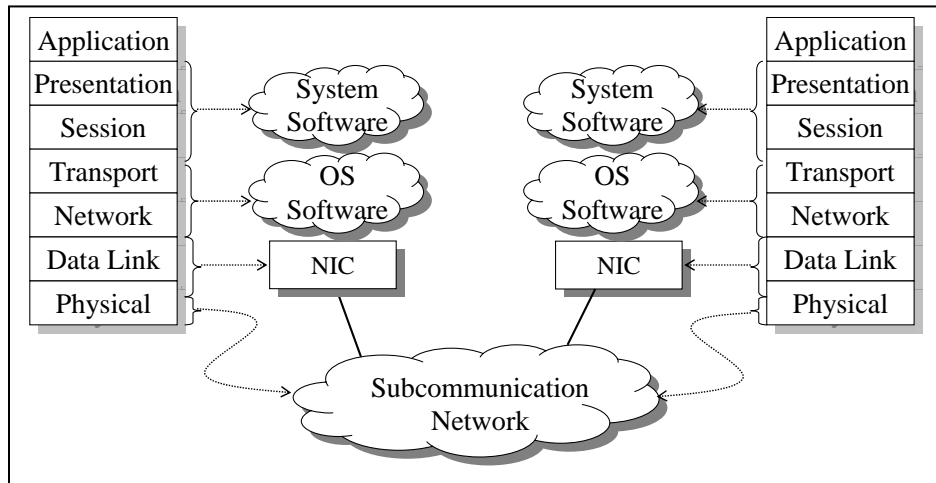


Figure 2-3: The ISO OSI Model

As you can see in Figure 2-3, there are 7 levels of API defined in the **ISO OSI** model:

- **Physical.** The physical layer is the lowest layer in the model. This layer defines how bytes are to be encoded and transmitted to other machines. The RS-232 asynchronous serial communication protocol (used to connect terminals, modems, and printers to computers) resembles a physical layer protocol, although it is not normally considered to be an OSI network protocol. The Ethernet carrier sensing and collision detection techniques are good examples of a physical layer network protocol. Part of this layer of protocol is implemented in the subcommunication network (the part that dictates how signals are transmitted and received), and part of it is in the NIC.
- **Data link.** This layer is built on top of the physical layer and, in the case of Ethernet network, is implemented in the NIC. (However, in the case of some older protocols like SLIP and PPP the data link layer is a software implementation using an RS-232 physical layer.² The data link layer defines a protocol that establishes *frames* of information that incorporate a header, a block of data, and a trailer – a frame is the data link name for a packet. A user of the data link layer can exchange frames with another host machine on a network. The data link layer NIC implementation will convert a frame into a stream of bytes when it sends a frame, and convert a stream of bytes into a frame when it receives them.
- **Network.** The network layer creates a very large address space of communication “endpoints” called an *internet address space*. This layer encouraged the development of the internet idea (*networks of networks*). Information is transmitted across the internet as *packets*. The network layer is usually implemented as a part of the OS. That is, a network layer packet is transmitted from one computer to another using data link layer frames. In order to use *the public Internet*, the computer uses the network (or higher) layered protocol for communication.
- **Transport.** The transport layer uses the network to provide various application interfaces to the services implemented by the network layer, including block, byte stream, and record stream communication. It is implemented, in part by the OS, with the remainder of the implementation being other system software (in a library or as other middleware). Computers that use the transport layer can exchange information as a stream of bytes. The transport layer converts a stream of bytes into packets when it transmits information using the network layer, and vice versa when it receives information.

² [Stevens, 1994], Chapter 2.

- **Session.** The session layer extends the transport layer functionality by applying specific interprocess communication strategies. For example, a network message protocol or a remote procedure protocol would be implemented at the session layer. The session layer is typically implemented as application libraries.
- **Presentation.** The presentation layer defines data abstraction and representation protocols. It is also implemented as library code. There are not many pure presentation layer protocols, other than things like the Sun RPC external data representation protocol.³
- **Application.** The application layer refers to the application software for the distributed computation. There are no standards for the application layer, since it is intended to apply to any domain. It exists in the model to illustrate where application programs fit into the layered architecture.

After seeing the role of the various protocols, and in reviewing Figure 2-3, it is apparent that while the crucial physical layer of the network is in the subcommunication network technology, much of the function of the network is in software. And since distributed computing depends on the perception of a seamless distribution of function across multiple computers (using the network), the behavior of distributed systems is greatly influenced by the character of the OS and other system software. We will focus on distributed programming and software in [Chapter 3](#), and the remainder of the book is about OS software. Before moving onto that discussion, we will discuss some other aspects of how a program can be distributed across execution engines – both within a computer, and across different computers.

2.2 Executing I/O Instructions

Almost every executing program needs to communicate with the external environment of the computer in order to do any useful work. For example, the program must obtain input data from a human, a sensor device, a storage device, or a communication device so that it will know which particular problem is to be solved. In the case of a sort algorithm, the external environment might supply the list of numbers to be sorted. Similarly, when the program has completed its execution on the input data, it will usually provide some output information to a human user, an actuator device, a storage device, or a communication device. The output is the product of executing the program on the input data. This basic idea for I/O is introduced in [Section 1.2](#) and is summarized in Figure 2-4 (a combination of the information in Figures 1-7 and 1-8).

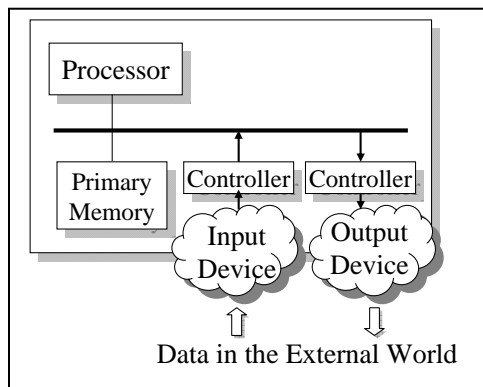


Figure 2-4: Input and Output Devices

All von Neumann computers are composed of a processor (or CPU), an executable memory, and multiple devices. Each of these 4 types of components is implemented as a separate unit of the hardware,

³ Srinivasan, R., “RFC 1832 – XDR: External Data Representation Standard,” IETF (Internet Engineering Task Force), Network Working Group, RFC 1832, August, 1995. See <http://www.faqs.org/rfcs/rfc1832.html>.

and each can perform its function relatively independent of – even at the same time as – the other functions. The processor executes almost every machine instruction, an important exception being the I/O instructions. Each I/O instruction is executed by a device. The device is started by an I/O command from the processor; it responds by performing the task specified by the command. For example, a device read command causes the device to provide data for the use of the processor, and a device write command causes information provided by the processor to be given to the device. Exactly what the device does to produce or consume data depends on the nature of the device: as we have seen, a write command to a NIC device causes it to send data to the subcommunication network, while a read command to a disk device cause it to copy data from its persistent storage back to the processor.

In particular, observe that once a device receives an I/O command, it can be busy reading input information or writing output information without requiring any assistance from, or interaction with, the processor – possibly for a relatively long time compared to the time it takes the processor to execute an instruction. That is, while most instructions execute on the processor in a tens of processor cycles, input and output instructions execute on separate devices requiring time that is orders of magnitude more than the time to execute a processor instruction.

Many decades ago, system designers became acutely aware of the program execution bottleneck introduced by I/O, particularly when devices that have mechanical motion in them, or ones that depend on human activity to complete the I/O (like typing on a keyboard), are involved. System designers observed that the program that caused the I/O operation would not be able to proceed for a relatively long time – namely the amount of time it would take for the device to complete the I/O instruction. The OS people came up with an interesting idea: why not let the program execute code that does not depend on the result of the I/O command during the time that the I/O command is being executed?

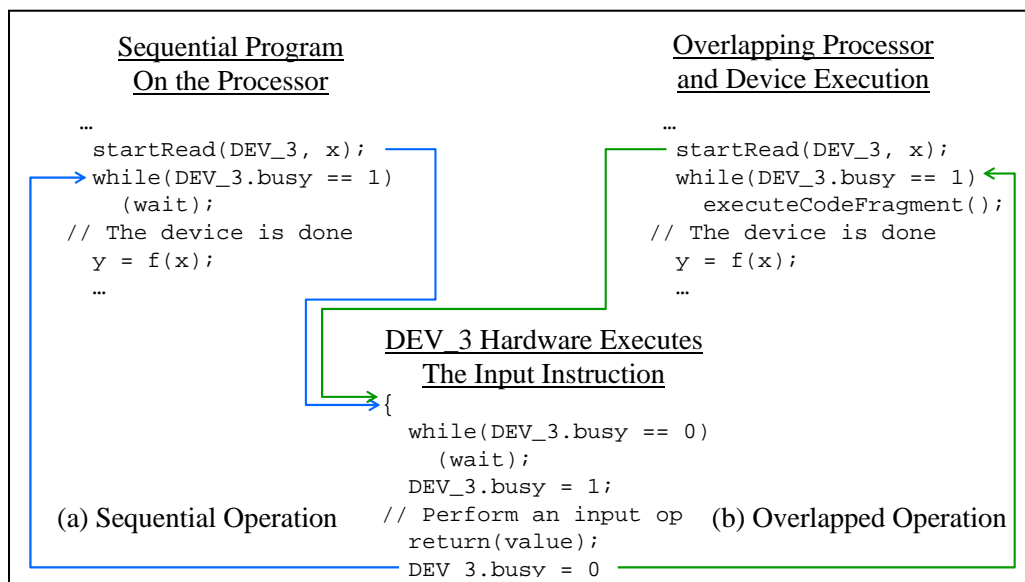


Figure 2-5: Sequential I/O Semantics

Here is the basic idea: assume that the `startRead()` machine instruction shown Figure 2-5, initiates the input instruction in the device, but then immediately enables the processor to execute another instruction after it has sent the device command to the device. Then the “normal,” safe way to execute the program is illustrated in Figure 2-5(a). As long as the device is busy on the I/O command, the processor executes a loop in which it waits for the input operation to complete (by constantly checking the device’s busy and/or done flags). Figure 2-5(b) suggests the alternative programming strategy that enables the program to be executing useful instructions on the processor at the same time that the device is in operation: after the `startRead()` instruction is executed, the processor doesn’t wait, but instead it

begins executing other code (`executeCodeFragment()` in the figure).⁴ In this approach, the programmer would be responsible for ensuring that the input operation has completed before using the value of `x` in the statement following the `while`-loop. If the program fails to fulfill this responsibility, then the program is likely to produce an incorrect result (since it might use an old value of `x` rather than the new value of `x` that would be read by the input statement). The upside of the second option is that two parts of the original sequential program can be executed at the same time, thereby decreasing the amount of time required to execute the program. The downside is that if the programmer is not careful, the program will produce incorrect (or at least unexpected) results.

2.3 Generalizing the Device Model (Sneaking Up On Distributed Systems)

In Section 1.2 and in Figure 2-4 we discussed how software executing on a processor can direct a device to perform an I/O instruction. Let's suppose that the device is a NIC, and that the write command has enough associated information to direct the NIC to send a packet to a particular remote computer. Then, at a low layer (NIC) control flow perspective, sending a message to a remote computer is not much different from writing data to a local device (see Figure 2-6, which is a redrawing of Figure 2-2). That is, the computer communicates with the NIC (and hence the remote computer) in much the same manner that it would with any other device: the processor determines when the device is ready to perform an I/O operation, and then it starts the device and waits for it to provide data on input, or to accept data on output.

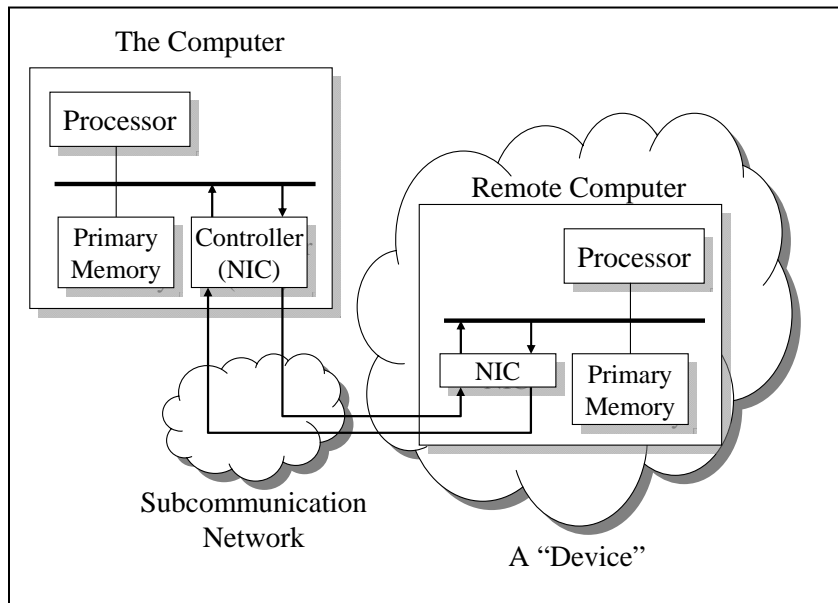


Figure 2-6: A Remote Computer as a Device

In Figure 2-7 we consider a pseudo code description of how the two computers can interact using the normal, safe device I/O paradigm (sequential execution across the local and remote computers) shown in Figure 2-5(a). Suppose that the local computer wants to transmit information to a remote computer. Let's assume that the NIC is able to execute a new command, `setAddr()`, that tells the NIC device where it should send data when it receives a `startWrite()` command. The first thing that the local computer does is to issue a `setAddr()` command. Since this is just a device command, the software waits for the NIC device to respond that it has completed executing the `setAddr()` command before proceeding.

⁴ There are various ways to implement this strategy in specific programming environments. You can find such techniques in textbooks such as Chapter 12 of [Stevens, 1992] or supplementary notes for this online book.

When the device is again ready to use, the local computer issues a `startWrite()` command to the NIC, and then again waits for the device to finish executing the command. Note that when the NIC has finished transmitting the information, it becomes idle, but that does not mean that the information has necessarily been received by the remote computer (only that it has been sent by the local computer NIC).

On the right side of Figure 2-7, the remote computer program that is going to receive the information from the local computer ultimately performs a `startRead()` command on its NIC device. The program then begins waiting for the device to complete the read operation, which could be a substantial amount of time, if the `startRead()` command was issued to the NIC before any other computer transmitted information to the given remote computer. On the other hand, another computer might send data to the remote computer's NIC before the application program issues the `startRead()` command. In this case, the NIC stores the data until the remote computer's software performs an input operation on the NIC. Once information has been received at the remote computer's NIC device, the device is able to finish its outstanding `startRead()`, and the processor detects this by observing that the busy flag gets reset to zero.

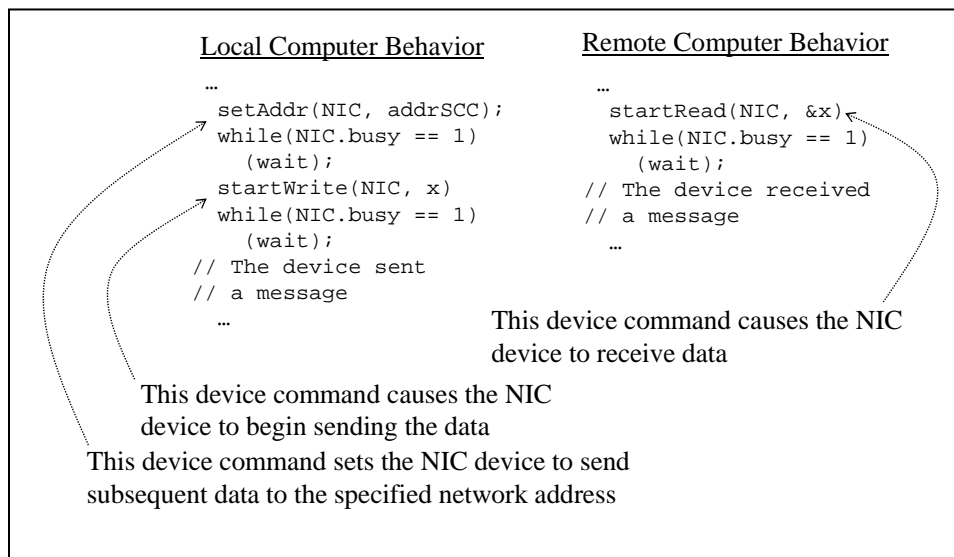


Figure 2-7: Computer-Computer Communication

This simplified example illustrates the normal, safe way to use a NIC to transfer information between machines; it is an exercise to write the pseudo code for overlapping processor activity with network communication. This discussion ignores the details of network communication (of which there are many) – we return to look at more details for programming NICs and networks in [Chapter 3](#).

2.4 Roles for SCCs in Distributed Systems

SCCs are much smaller than typical computers in a distributed system (personal computers, workstations, server-class machines, mainframes, clusters, and multiprocessors), and also tend to be specialized to perform a particular task or to handle a small number of tasks in a particular domain. When considering the role of an SCC in a distributed system, the SCC can either *provide* a service to other computers in the distributed system, that is, it plays the role of a *server*; or it might play the opposite role of *using* services provided by other computers in the system, that is it behaves as a *client*. We will consider these two roles separately.

2.4.1 Providing Services to Other Computers

SCCs are generally limited by the amount of resources that can be configured into them. This suggests that when an SCC provides services to other computers in the distributed system, the SCC is acting as an

intelligent interface to some other component – perhaps as a sensor for instrumentation, or as an actuator to control parts of some external machine (like a robot or a rocket ship). There can be other applications in which the SCC provides service, but certainly the sensor and actuator controller is a prominent and representative use. We will focus on this example to describe how SCCs can provide services to other computers.

Some devices require constant attention to accomplish a given task (commonly referred to as “dumb” devices), while “smart” devices are able to handle complex tasks without interacting with the software during task execution. While it may be possible for both devices to ultimately perform the same task, the smart device is directed to perform the task using a single high-level command. On the other hand, a dumb device tends not to incorporate algorithms to perform tasks, instead it performs its work under the direction of a detailed set of low-level commands. A simple example demonstrates this distinction between smart and dumb devices: suppose that we wanted to draw a circle on an output device (such as a graphic printer or video screen): some graphic devices can only draw points and lines; to draw a circle, the device must be instructed to draw a collection of small line segments that are connected to form a polygon. The program could draw 3 line segments to form a 3-sided polygon (an equilateral triangle) to represent a circle, 4 line segments (a square), or 90 line segments to form a polygon with 90 sides. Obviously the 90-line approximation of the circle looks more realistic than does a triangle, square, pentagon, hexagon, and so on. Ultimately the number of sides that should be incorporated into approximating a circle depends on the desired graphic quality of the circle, and on the graphic device’s resolution.

A smart(er) graphic device can respond to a single command to draw a circle (with the center at a specified graphic location, and with a given radius). The smart device determines the number of sides on the polygon, and the specification of each of the lines that form the sides of the polygon. The program that intends to draw a circle on the dumb graphic printer must decide how many sides the polygon should have, the slope, length, and placement of each line segment, and then it must issue N commands to draw the N lines. The smart graphic printer *incorporates its own algorithm* for drawing a circle, that is, the algorithm determines the number, length and orientation of the line segments in the polygon, whereas the dumb graphic printer leaves the responsibility for that algorithm to the program that directs the device; this is the rationale for the “smart” and “dumb” designations.

Let’s generalize this idea of designing smart devices into one of distributing parts of an algorithm across the processor and devices. If the device is so simple that it can only print a dot on a Cartesian space (X-Y plane), then the program will need to perform computations to determine which points in the display space need to be darkened in order to form a letter, draw a line, or draw a circle. But if the device is smart enough to determine which dot pattern to print to form letters, then programs need only tell the device to print a designated character (perhaps in a designated font). If the device does not incorporate algorithms to enable it to draw a line in the X-Y space, then the program will be required to determine which dots needed to be printed in black in order to draw an approximation of a line, say from point (53, 12) to (56, 94). And so on. In any case, some part of the computer (the program executing on the processor or the device) has to determine which dots to blacken for each of the print operations: simple devices leave that computation to the application program, while smart devices are able to execute part of the processor’s algorithm on their own. The OS community thinks of the smart device scenario as being one in which *work is distributed across different hardware units* – in this case across the processor and devices. This basic idea is fundamental to the design of distributed systems.

SCCs can be used to add “intelligence” to devices in a distributed system. Imagine that we wish to build a (large) computer system that controls the heating and cooling in a large cruise ship with, say 1,500 rooms. We assume that the ship is large enough that some rooms might have to be cooled at the same time that others are being heated. Suppose each room contains a digital thermometer that the central computer can read to determine if it needs to heat or cool the associated room. Clearly the large computer needs to periodically read the temperature in each room, and decide if it needs to start the corresponding heater or cooler for the room. Simplistically, we expect that there is a loop in the main computer that implements a code fragment similar to the following:

```
...
/* The temperature maintenance loop
 * thermometer[i] is the device in room #i
 */
for(i=0; i<NUMBER_OF_ROOMS; i++) {
    startRead(thermometer[i], &currentTemp, ...);
}
```

```

while(thermometer[i].busy == 1)
    /*(waiting)*/;
/* We have a current temperature for room #i */
if(currentTemp >= maxTemp) {
    /* Turn on the air conditioning */
    ...
} else
if(currentTemp <= minTemp) {
    /* Turn on the heater */
    ...
}
...
}
...

```

The quality of heating and cooling service provided by the computer depends on the frequency with which the loop is executed. But executing it too often will just waste processor cycles. Notice that the frequency might change with the seasons, and with the route that the cruise ship takes (the Arctic cruise versus the Seattle-San Francisco cruise).

The code fragment could be simplified considerably if the simple digital thermometer device were smarter, that is, so that incorporated some of the algorithm that the computer has to implement to check threshold values. Suppose a *very simple* SCC was associated with each digital thermometer, as show in Figure 2-8. This SCC would be dedicated to executing a portion of the algorithm that was previously implemented in the large computer, that is, the large computer is now the main machine in a distributed system, and there is one SCC associated with each thermometer.

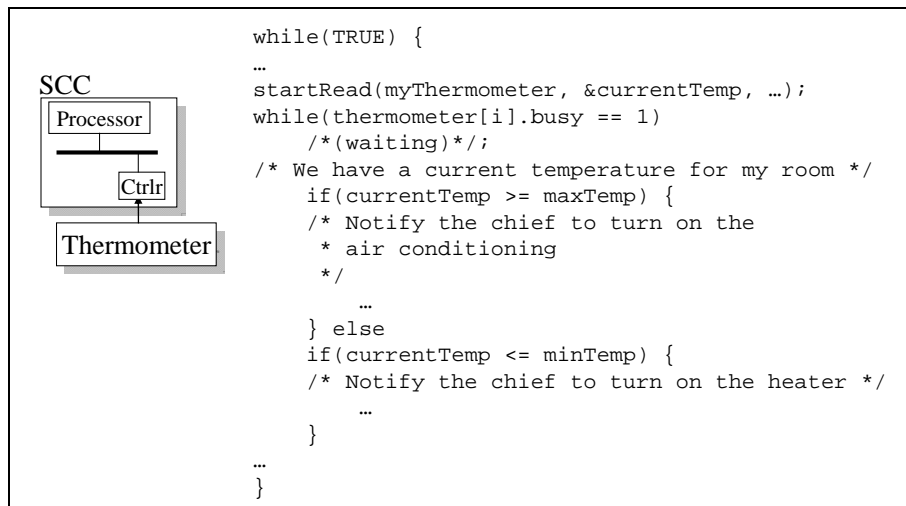


Figure 2-8: Distributing Intelligence to the Thermometer SCC

Figure 2-9 shows the relationship among a thermometer, its associated SCC, and the large computer (in the cruise ship example there would be 1500 SCCs connected to the large computer, unless the SCC managed, say 5, thermometers, thereby reducing the number of SCCs to 300). The configuration uses the network (in the manner suggested by Figure 2-6) to connect the SCC-thermometer nodes to the large computer. While the SCC incorporates the algorithm to directly manage the thermometer, but the parameters for its behavior are determined by the large computer. The SCC only transmits data to the large computer when the temperature falls below a low threshold value, or rises above a high threshold value, and it accepts the following configuration commands from the large computer:

- Set the high threshold value
- Set the low threshold value

- Set the mode of the thermometer to report temperatures in either Celsius or Fahrenheit scale
- Suspend the thermometer activity
- Activate the thermometer

The large computer initializes (the SCC and the) device by setting the high and low threshold values, and then by activating it to send values to the large computer. The suspend command is used to temporarily disable the thermometer so it does not send data to the computer, and the mode command directs the SCC to report the temperature in either Celsius or Fahrenheit degrees.

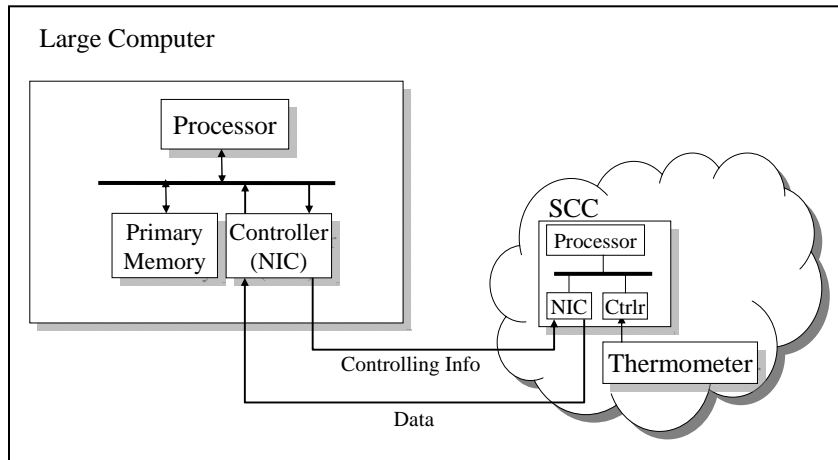


Figure 2-9: A Device Controlled by an SCC

Let's consider in a little more detail how the SCC and the large computer might interact with one another:

```
while(TRUE) {
  /* Check the Network for info form the large computer */
  startRead(NIC, &cmd, ...);
  while(NIC.busy == 1)
    /*(waiting)*/;
  switch(cmd) { // Process the command from the large computer
  case HI_THRESHOLD_TEMP: // Set the high threshold temp
    ...
    break;
  case LO_THRESHOLD_TEMP: // Set the Low threshold temp
    ...
    break;
  case SET_MODE: // Set the mode to C or F
    ...
    break;
  case ACTIVATE: // Start measuring the temperature
    activated = TRUE;
    ...
    break;
  case SUSPEND: // Pause measuring the temperature
    activated = FALSE;
    ...
    break;
  default:
    break;
  };
};
```

```

if(!activated)
    break;    // Skip temperature measurements
/* Check the temperature */
startRead(myThermometer, &currentTemp, ...);
while(thermometer[i].busy == 1)
    /*(waiting)*/;
/* We have a current temperature for my room */
if(currentTemp >= maxTemp) {
    /* Notify the chief to turn on the
     * air conditioning
     */
    startWrite(NIC, ...);
    ...
} else
if(currentTemp <= minTemp) {
    /* Notify the chief to turn on the heater */
    startWrite(NIC, ...);
    ...
    ...
}
...
}

```

First, the SCC now includes code to accept commands from the large computer. It checks the NIC to see if there is any information – if so it uses the `switch` statement to parse and executed the command. Next, is the code to manage the thermometer – this is only executed if the large computer has activated this SCC-thermometer. Finally, the SCC retains the code introduced in Figure 2-8 to manage the thermometer.

Here is a brief description of the pseudo code description that the large computer uses to interact with the SCC:

```

/* Initialize SCC-thermometers
 * Send commands to all SCCs (use broadcast if available)
 */
...
/* Periodically check the NIC to see if any SCC is trying
 * to send any information to me (the large computer)
 */
startRead(NIC, ...);
if(NIC.done == 1) {
    /* We have a message. Parse it to determine:
     * - which SCC transmitted the device
     * - the reason for the message - normal or exception
     * - determine the response to the message, if any
     * - execute the response, if any
     */
    ...
}
...

```

This is an example of a SCC being added to directly control the original “dumb” device, thereby converting it to a “smart” device, it is possible to add other features to the SCC so that the device appears to be even more intelligent.

This method of using small computers is the most traditional use, although not necessarily in conjunction with networks. In the field of embedded computers, this is the usual way that a computer is used to implement functionality in a machine that is not necessarily able to execute algorithms. Close to the computer domain, device controllers themselves are often small computers. For example a contemporary serial port is usually implemented with a specialized small computer called a universal asynchronous receiver/transmitter (UART) that is able to implement parameterized algorithms for

transmitting and receiving bytes over a wires. A more impressive example is the logic on a contemporary hard disk – in this case, we are referring to electronics that are part of the device rather than part of the controller. The algorithms necessary to actually operate the disk drive under a given disk protocol (like SCSI) are often implemented using a small computer rather than discrete logical components.

2.4.2 Using Services Provided by Other Computers

The trend toward portable electronic devices has stimulated a usage paradigm in which the SCC behaves more like a notebook computer than like an embedded system. They request services from larger machines on the network (rather than vice versa as described in the previous section). For example many cell phones use a broadband data network to transmit and receive data from/to the cell phone, thereby enabling the cell phone to be used to surf the web and to retrieve a copy of email that has been delivered to larger machines. In general, when the SCC assumes this role in which it requests services from other machines (rather than supplying services to other machines), the SCC behavior resembles more conventional machines such as notebook and desktop computers.

Today, machines in a distributed system are often characterized as being either a “client” machine or a “server” machine. This characterization is informally used to differentiate between large, shared machines and smaller machines dedicated to one task or to supporting one human user. In reality the terminology stems from the roles that the software in these computers is organized: in some case the software requests service from another computer, while in other cases it provides service to other computers. The two computers are said to *play the roles* of **clients** and **servers**, where client software requests service from server software. The software, and hence the machine, change roles, depending on the nature of the distributed task, and which part of the task the computer is currently fulfilling. That is, whether a machine is a client or a server depends only on the current behavior of the software, not on the size or type of the hardware. Historically large, shared computers tend run software that provides services, so they have become known as “server machines.” Similarly, historically small computers, such as PCs, request services from large machines, so they have come to be known as “client machines.” Unless explicitly mentioned in the discussion, we will bow to the popular usage of these terms and often refer to large machines in a distributed system as “server machines,” and small computers as “client machines.”

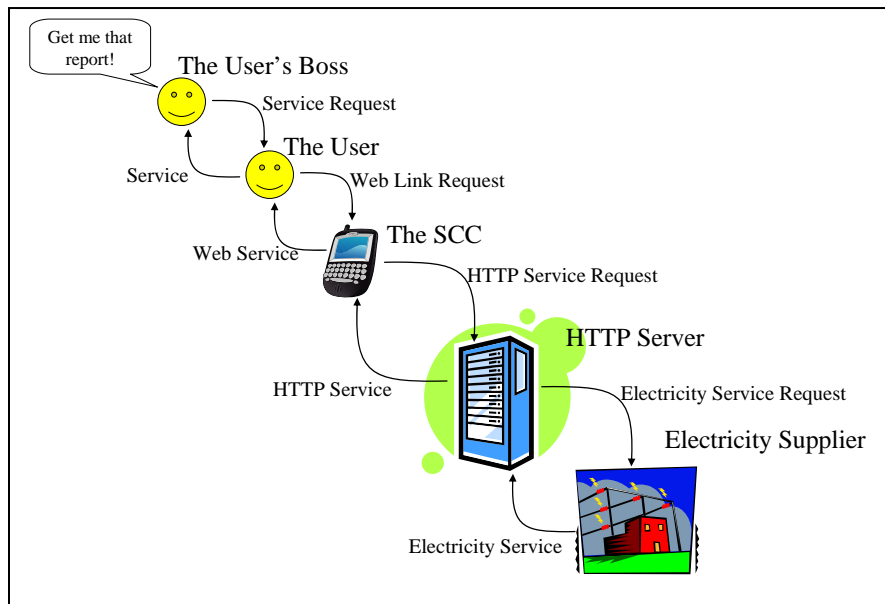


Figure 2-10: Of Clients and Servers

Client machines act as servers in the sense that they provide services to their environment. As suggested by Figure 2-10, if the client machine is a personal computer, cell phone, or PDA, the machine’s human user issues the directives to the SCC. (Figure 2-10 carries the client-server relationship to the

extreme by showing that even human users are servers to other humans such as their boss, and that even an HTTP server is a client to the electric utility company in the sense that it requests electricity from the electricity supplier.) Once the user has requested a service from the SCC, the SCC will use its own programs and resources to accomplish the action, or it will temporarily assume the role of a client to request help from servers in the distributed system. For example, if a handheld SCC is running a web browser program, when the user clicks on a link, the SCC temporarily suspends its role of user server and assumes the role of an HTTP client that requests a copy of an HTML file from an HTTP server.

The role of server software is to wait passively for requests for service, to perform that service, and then to resume waiting passively for other requests. The role of client software is to implement an independent algorithm, and to delegate some of the work to servers in the distributed system by requesting that they perform certain tasks in the overall computation.

Computers that execute software which *requests* information and/or services are called **client** computers, and those that *provide* information and/or services are called **server** computers.

In the server role, the difference between an SCC and, say, a notebook, is that SCCs are caused by the nature of their physical size, which tends to limit the amount of resources that can be configured into the SCC. This suggests that the SCC will tend to make more frequent client requests for the use of resources on server machines in the distributed system than would a larger client machine. For example, many SCCs do not include a hard disk drive, although they are likely to incorporate storage devices of limited capacity (compared to storage capacities of devices on desktop computers), such as flash drives. Like other machines in a distributed system, they will store and retrieve many of their files on a larger file server machine; but unlike larger client-style machines, they are likely to access the file server more often than other file clients since they are unable to store copies of very many files in their storage devices. SCCs are designed in an attempt to optimize their performance under the assumption that they will make more frequent use of servers than would a conventional notebook or desktop computer.

Thus, when an SCC is generally a client machine in a distributed system, it generally behaves much like conventional notebook and desktop computers. Once the SCC is incorporated into a distributed system, it is obliged to use a network with a communications framework (such as an instance of the OSI model) that is universally used in that distributed system. The SCC operating system provides the mechanism for the foundation of the distributed system coordination, and other system software is responsible for implementing the higher layer protocols of the communication. Before we can begin to look at OS details in [Chapter 4](#), we will consider programming models for distributed systems in the next chapter.

2.5 Summary

All SCCs exist as machines in a distributed system. The SCC might be so simple that it can be implemented by a very simple program that continuously runs a simple loop, for example to read a thermometer, check the reading for threshold values, and then either transmit the reading or ignore it. Such uses of small communicating computers enables some interesting design possibilities for the SCC: should the SCC be a very simple computer that is dedicated to a single task, and has very little work to do. Or should the SCC itself incorporate hardware features found in larger computers, such as interrupt driven devices, operating modes for the CPU, and so on? Should the SCC focus on supporting software that handles a dedicated task, such as monitoring a sensor, or should it be expected to handle multiple tasks such as reading a set of sensors and controlling a set of actuators, for example, to guide the trajectory of a flying object? In our study of SCCs, we will determine that there are many different types of SCCs where some are modest machines that only handle a dedicated task, while others are expected to perform substantial information processing work.

Independent of the amount of responsibility delegated to the SCC, since it is a communicating small computer, there is an implicit assumption that it communicates with a remote machine. That is, in this book *every SCC is a component in a distributed system that executes distributed programs* on a combination of the SCC and one or more other machines. Studying SCC operating system is distinguished

from studying generic operating systems since every SCC must be designed to operate in the context of a distributed system.

Sometimes the SCC is assigned work to reduce the amount of processing that a host machine would otherwise have to perform, and sometimes (as in the thermometer controller case) work is assigned to the SCC to reduce the amount of information that needs to be exchanged between the two computers. Today, SCCs are sometimes packaged with other components so that they meet other, nontechnical needs; for example the SCC may be part of a cell phone that exchanges files with a server. You can expect that even these “loosely coupled” systems will evolve over time so that the SCC and other machines will share information using events, message, and objects in addition to information contained in files.

2.6 References

1. Stevens, W. Richard, *Advanced Programming in the UNIX® Environment*, Addison Wesley Professional Computing Series, 1992.
2. Stevens, W. Richard, *TCP/IP Illustrated, Volume 1: The Protocols*, Addison Wesley Professional Computing Series, 1994.
3. Tanenbaum, Andrew S., *Distributed Operating Systems*, Prentice Hall, 1995.