

## **Preface**

When thinking of something to do research on, I wanted to find something that would not only interest me, but also be something I could use in fifteen years possibly. I have always thought it was a waste to have twenty computers in one room, three people each at a workstation, and have to sit around while the computer swaps to disk in order to run Microsoft Word, Access, and Excel simultaneously. I always wondered if it would be possible to use resources from other nodes in the network to help me do the jobs I was trying to accomplish. The idea of having a distributed system is almost overwhelming to me. The design intrigues me and it's much simpler in rhetoric, but the idea is so new that I haven't heard whether or not it would be a successful "venture." Instead of attempting to write 400 pages on the idea of distributed operating systems, I chose to limit my research to message passing in message-oriented interprocess communication of a distributed operating system. The first few pages explain both the concept of a distributed system and a distributed operating system, in order to lay a small portion of the background for the reader's understanding of interprocess communication.

## **Distributed System**

A distributed system is a collection of independent computers and some communication facility for exchanging messages (Maekawa 177). A distributed system can also be an interconnected collection of autonomous processes or processors. These computers, processes, and processors are referred to as the nodes

of the distributed system. There are five key reasons why distributed computer systems are being chosen as the information systems setup in many workplaces:

- (1) The need for **information exchange**. In the sixties, people began to see the need for easy access to information that was located on other computers in their workplace, hence, the development of WANs (wide-area-networks). All they needed now was a way to exchange electronic information between these interconnected computers.
- (2) The need for **resource sharing**. It's easy to equip each user in an organization with a personal computer but it becomes more costly when you buy everyone other peripherals such as printers, hard drives, and backup storage. Sometimes, it's just not feasible to spend a huge sum of money on a mainframe computer when one can buy several workstations at a slight fraction of the price, and then implement these computers as a distributed system. If the capacity of your organization's has been reached, you can always add more file servers, printers, or computers. If the capacity of your mainframe has been reached, your only option is to purchase another more expensive mainframe that meets your needs.
- (3) The need for **increased reliability by replication**. Distributed systems are more fault tolerant than centralized systems because they have a partial-failure property. When one node of a distributed system fails, another node can take over the tasked of the failed node, eliminating bottlenecks in the

system. If you're in a standalone mainframe or a centralized computer, there is no possibility of continuing the operation because the failure affects the entire system.

- (4) The need for increased performance through **parallel processing**. Parallel processing creates new opportunities for splitting highly intensive computations over many processors, therefore decreasing the turnaround time. Workstations can benefit from this technology when they are overloaded with processes and tasks; they can send them to another workstation to finish.
- (5) The need for **simplification of design** through specialization. Highly functional computer systems are very complicated to design and maintain. This can be simplified by splitting the system into modules that communicate with each other and each share a part of the total functionality of the computer (Tel 3). An example of a network based distributed system is located in Figure (1).

### **Distributed Operating System**

The communication facility that organizes the exchange of messages with other independent computers in the distributed system is referred to as the operating system. A distributed operating system should:

- (1) Control network resource allocation to allow their use in the most effective way;

- (2) Provide the user with a "virtual computer" that serves as a high-level programming environment;
- (3) Hide the distribution of the resources;
- (4) Provide mechanisms for protecting system resources against accessing by unauthorized users; and
- (5) To provide secure communication (Goscinski 7).

A distributed operating system should look like a centralized operating system but should actually run on multiple processing units which appear transparent to the user and also show the system as a uniprocessor, instead of many computers and their processors joined together by a network. The distributed operating system is intended to support application programs whose instructions may be distributed for execution on different machines. It does much more than just sharing files, though; it shares as specific as records in a high level programming language, strings, or other simple data structures. It also has to control resources such as allowing distributed computations to synchronize before they write their findings to a shared portion of hard disk or wherever they choose (Nutt 24).

### **Message Passing Primitives**

Message passing was adopted for distributed computer systems in the late 1970s. It is a programming primitive for communication among the nodes where the basic programming constructs are send and receive. These constructs coordinate message-passing actions between two processes (Coulouris 34). They also are used

to synchronize processes. A message can be defined as a "block of unspecified information formatted by a sending process in such a manner that it is meaningful to the receiving process" (Nutt 109). It's a typed collection of data objects consisting of a fixed size header and a variable length body, which can be managed by a process, and then delivered to its destination. Most of the times, the content of the message is determined by the sender. A simple message header with data is displayed in Figure (2).

Messages have two options: to be structured or unstructured. When unstructured messages are sent to user processes, there exists problems because some messages have to be interpreted by the distributed operating system's kernel because they have to be translated to be meaningful to other processes. This causes system overhead because messages have to be encapsulated and decapsulated from structured form into unstructured form. This inefficiency increases the cost of communication.

Message oriented communication can be likened to the client-server model (Figure 3). One process called a client, sends a message (request) to another process called a receiver. It waits for a reply, or keeps running. A message is sent and received using the semantics below (Goscinski 137).

**send** *expression\_list* **to** *destination\_identifier*

the message contains the values of the expressions in *expression\_list* at the time **send** is executed. The user can control a destination of a message with *destination\_identifier* (one destination or many destinations).

**receive** *variable\_list* **from** *source\_identifier*

*variable\_list* is a list of variables. The user can control where the message came from with *source\_identifier*.

Figure 3 shows that the client process issues a send (server, message), and the server process issues a receive (process id, buffer). The client has to specify the destination address (server) and the message, and the server process tells who they're getting the message from, and allocates a buffer to store the incoming message (Goscinski 138). No connection is established, therefore, no disconnect is necessary. It is important to note that the server also uses the send command to send messages back to the original client; he isn't bound to just receiving messages. If it is necessary, one can use a guarded receive that contains a guard or boolean condition or the receiving process can specify exactly who it expects to selectively receive the message from. These primitives or concepts are shown below (Goscinski 138).

<p><b>Guarded Receive</b>  <b>receive</b> <i>variable_list</i> <b>from</b> <i>source_identifier</i> <b>when</b> B  permits only receipt of messages if B is true.</p> <p><b>Selective Receive</b>  <b>select</b>      <b>receive</b> <i>variable_list</i> <b>from</b> <i>source_id1</i>      <b>or receive</b> <i>variable_list</i> <b>from</b> <i>source_id2</i>      <b>or receive</b> <i>variable_list</i> <b>from</b> <i>source_id3</i>  <b>end</b></p>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

One way of transmitting messages from one process to another is by copying the body of the message from the memory of the sending process straight into the memory of the receiving process (Nutt 109). Sometimes, the receiving process isn't ready for the information from the sender. In that instance, the process would like

the operating system to stick the message in a "mailbox", whose data type is a queue, for later reception. This scenario is illustrated in Figure 4. Two copies are necessary for this exchange: one copy from the sender's memory to the receiver's mailbox, and another copy from the mailbox to the receiver's memory when he asks for the message. Two more system calls are necessary to transmit and receive the messages; you use a send call to place the message into the receiver's mailbox and a receive call for the receiver to retrieve his messages from the mailbox into his memory (Nutt 112).

### **Blocking Versus Nonblocking Primitives**

A primitive has nonblocking semantics if it never delays the invoker when it executes, or else it's said to be blocking. To have a nonblocking setup, you must have a buffer, usually in the form of a queue. A nonblocking primitive has several functions:

- (1) **Send** returns the control back to the user program as soon as the message has been queued;
- (2) After a message is transmitted or copied safely to a queue, the user program is interrupted to let it know that the buffer may be reused.
- (3) The corresponding **receive** primitive says that it is ready to receive another message and opens a buffer in which to place the incoming message.
- (4) When a message arrives, the program is notified by interrupt (Goscinski 140).

The biggest advantage of nonblocking primitives is that they provide so much flexibility. These primitives are useful for real-time applications (Goscinski 140). If you are in a fighter aircraft, you don't want a message that says that the cabin pressure is too high to be blocked; it's critical that the message be delivered immediately. There are a couple of disadvantages to nonblocking primitives: if a buffer is full, a process must be blocked which defeats the purpose of using this primitive in the first place, and this primitive also makes programming difficult and tricky. It's not too easy to write timing dependent programs.

Blocking primitives are an easy way to combine data transfer and synchronization. If a **send** process is blocked, it must wait until the receiver actually receives the message. The receiver is also blocked until the sender has executed the **send** request. Sending and receiving processes are then synchronized. There are a couple of functions of blocking primitives:

- (1) To ensure reliable blocking, the **send** command doesn't return control back to itself until the message has been sent to the receiver and an acknowledgment received.
- (2) **Receive** doesn't return control until a message has been placed into its buffer.

The main drawback to blocking primitives is that they don't allow parallelism. A process that is blocked in **send** or **receive** mode can't do anything worthwhile (Goscinski 141).

### Buffered versus Unbuffered Primitives

With a system that uses buffered messaging, the message passing is said to be asynchronous, that is, the sender has a no-wait **send**. In most systems, the sender is allowed to have multiple **sends** but is bounded by a certain limit which is specified in the operating system kernel. Most often, the user is given a system call **create buffer**, which creates the kernel buffer of a size specified by the user, referred to in this paper as a **mailbox** or **port**. The sender sends a message to a receiver's mailbox or port, where it's buffered until requested by the receiver. They have a few drawbacks as well: They are more complex to implement. They create protection problems. If a process that owns a port dies, it causes huge event problems (Goscinski 142).

If no buffering is to be used, processes must be synchronized in some way in order to exchange messages. This synchronization can either be called a local rendezvous or a remote rendezvous. This deals more with the specifics of synchronization than message passing so I will briefly define both. A local and remote rendezvous are used for "bi-directional communication between two processes in the same program executing in the same address space" (Goscinski 143). These synchronization methods are further explained in Goscinski's book on pages 143-146.

### Unreliable versus Reliable Primitives

In local area networks, many different horrendous events can happen possibly causing communication system failure by a requesting message being lost in the network, a response message being lost or delayed while responding, or the responding node dying or being unable to be contacted. Sometimes, messages can also be duplicated or delivered out of order. These effects stated above were due to unreliable primitives. An unreliable **send** puts a message on the network and does not guarantee delivery of anything. With reliable messaging, the **send** primitive handles lost messages by doing internal retransmissions, and acknowledgments based on timeouts. This means that when the **send** primitive terminates, it knows that its message was both received and acknowledged (Goscinski 147).

The difference between reliable and unreliable **receive** is that the reliable receive sends acknowledgments to confirm that it received a message; an unreliable receive does not confirm receipt. A fellow named Saltzer proposed an idea in 1984 which stated that it would suffice to take care of the recovery at a process level as opposed to doing it at a lower level. Two way communication is a must in this reliable message passing model. If a client requests data, the server sends a response using the **send** primitive. The client then has to setup a **receive** primitive in order to receive a message back from the server.

## Structured Forms of Message Passing

Structured forms of message passing based communication are required if you are seeking a high-performance communication among distributed operating systems. When you have a standard or structure for primitives, this limits confusion for the operating system. This structure for message passing is helped by distinguishing between requests and replies and allowing bi-directional message flow (Goscinski 150). These primitives are outlined below.

<p><b>"send</b> sends requests and gets replies; it combines a previous client's <b>send</b> to the server with a <b>receive</b> to get the server's reply.</p> <p><b>get_request</b> is done by the receivers (servers) to acquire messages containing work for them to do.</p> <p><b>send_reply</b> the receiver (server) uses this primitive to send a reply after completing the work" (Goscinski 150).</p>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## Direct and Indirect Communication

Maybe an obvious question someone might ask is "Where do the messages go." The sender can either communicate directly by designating a fixed destination process or else communicate indirectly by using a fixed location for receipt of a message. Direct communication requires that processes specifically name the

recipient or sender of the communication. The **send** and **receive** concepts are shown below.

<p><b>send</b> (P, message) send a message to process P.</p> <p><b>receive</b> (Q, message) receive a message from process Q.</p>
-------------------------------------------------------------------------------------------------------------------------------------------

In this model, a link is established between every pair of processes that want to communicate, and the processes only have to know each other's identity to communicate with each other. A link is associated with exactly two processes, and between each pair of communicating processes, there exists exactly one link. The link must be bi-directional. There is a symmetry in this mode of addressing because both the sender and receiver have to name each other in order to exchange messages.

Another way to communicate is indirectly by the usage of ports. A port can be hypothetically viewed as a protected kernel object where messages can be sent and also retrieved. Processes can have ownership, send, and receive rights on a port (Goscinski 155). The concept by which ports communicate with each other is shown by the following.

<p><b>receive</b> (B, message) receive a message from port B</p>
----------------------------------------------------------------------

A communication path between ports has the following attributes:

- (1) can be associated with more than two processes
- (2) there could be different paths between each communicating process, where each corresponds with one port
- (3) can be uni-directional or bi-directional (Goscinski 155)

There are queues at each port where a process can refer to that port by a logical name. These queues follow FIFO order unless there is an emergency message that needs special attention. A port can be created by a process which owns the port. This process also has receive access to the port. To be able to destroy a port, a process must both own and have receive access to it. If the process who owns and has receive access to it dies, the port should die as well and the processes who were not the owner but had access rights, should be notified. There is a finite length for message queues attached to ports in order to control the flow of data and keep the queue from receiving more data than the system can absorb (Goscinski 157).

### **Getting Data from a Process**

There are two options for passing data from one process to another: either pass data by value through the exchange of messages, or pass data by reference. If you pass the data by value, a copy of the data is made and passed. This involves huge overhead and data copying costs. If you pass the data by reference, it only requires that you send an address by which the other process can share access to a

specific memory area or entire address space. These small pointers used in reference do not take up much room and are much cheaper than being copied more than once. The drawback to passing data by reference is that the programming task becomes more difficult because of issues with both virtual memory management and interprocess communication (Goscinski 158).

### **Conclusion**

Distributed operating systems, because of the variety of hardware it has to keep track of, become very complicated to implement because there is no set standard for operation. There are primitives or solid concepts by which to model different schemes of messaging. It's not as easy to share data between processes as it is in centralized systems, because there is no shared memory. Messaging needs can change depending on the needs of the company wanting to use a distributed operating system.

### Appendix

Figure 1

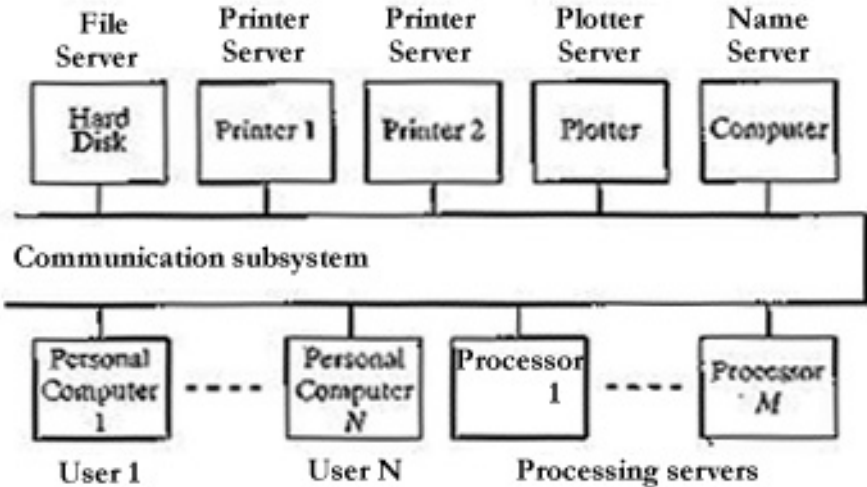
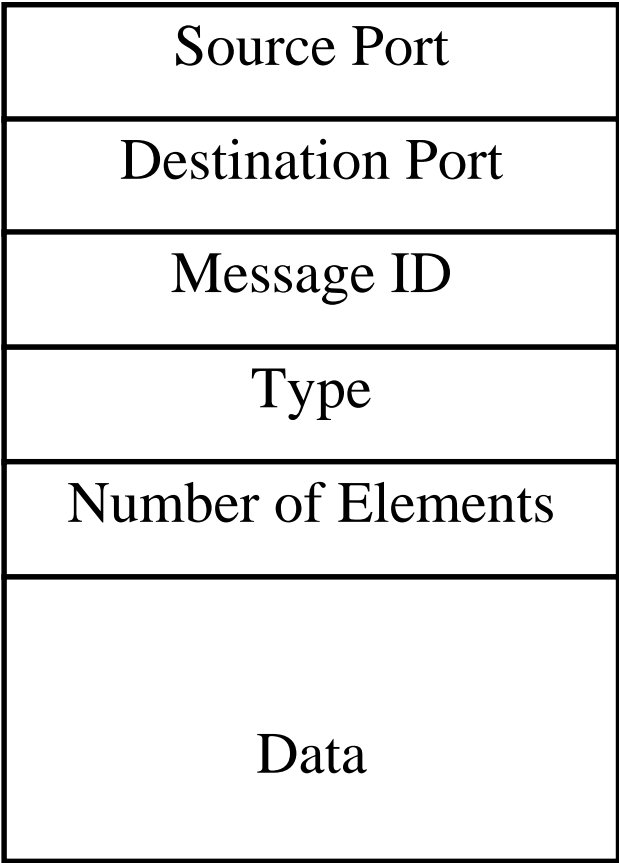
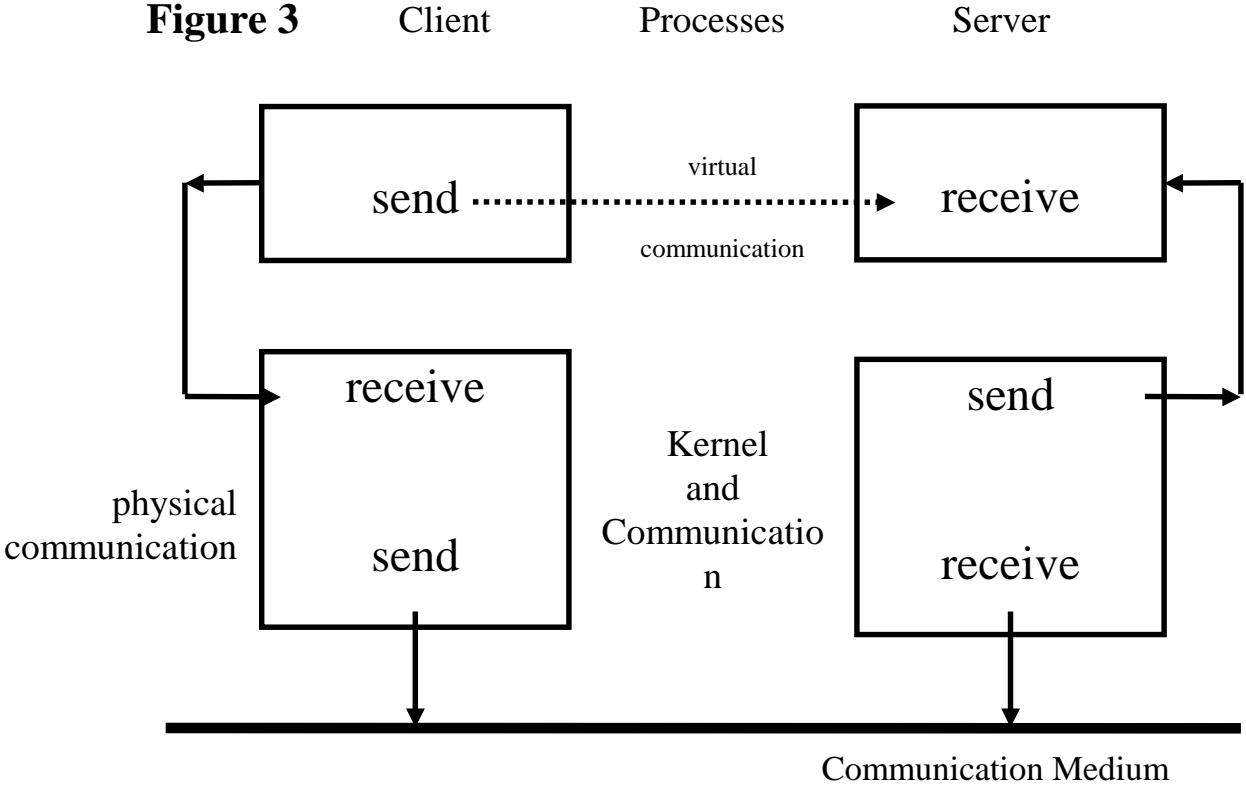


Figure 2



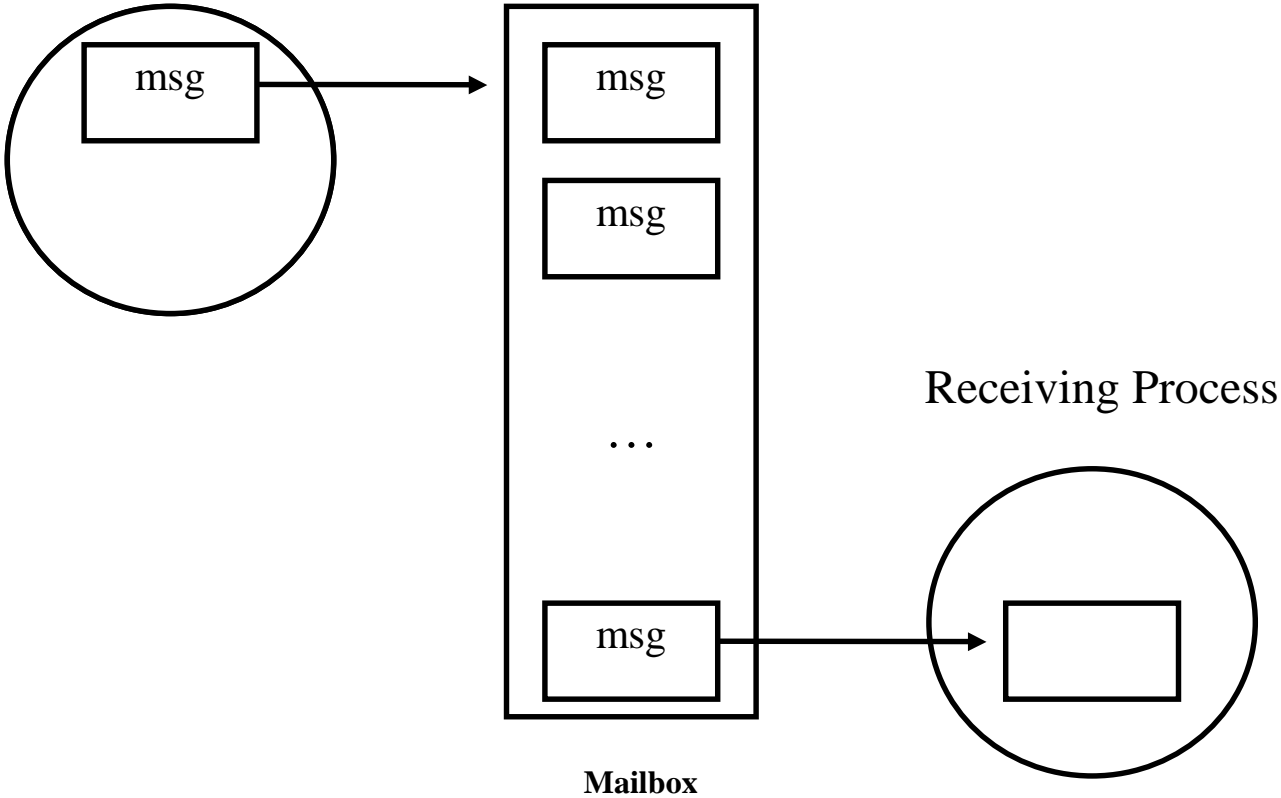
Simple message format

**Figure 3**



**Figure 4**

Sending Process



## Works Cited

Coulouris, George. Distributed Systems. Addison-Wesley Publishing Company, 1994.

Fortier, Paul J. Design of Distributed Operating Systems. New York, NY: McGraw-Hill, Inc., 1986.

Goscinski, Andrzej. Distributed Operating Systems: The Logical Design. Addison-Wesley Publishing Company, 1991.

Maekawa, Mamoru, et al. Operating Systems: Advanced Concepts. Menlo Park, CA: Benjamin/Cummings Publishing Company, Inc., 1987.

Nutt, Gary J. Centralized and Distributed Operating Systems. Englewood Cliffs, NJ: Prentice, 1992.

Tel, Gerald. Introduction to Distributed Algorithms. New York, NY: Cambridge University Press, 1994.

Bibliography

amoeba-support@cs.vu.nl. "The Amoeba Distributed Operating System." April

1996. <http://www.am.cs.vu.nl/amoeba.html/> (March 20, 1997).

Walker, Janice. "MLA-Style Citations of Electronic Sources." August 1996.

<http://www.cas.usf.edu/english/walker/mla.html/> (March 24, 1997).