\

Experiences Implementing SMTP with PowerPlant

Christopher Haupt, CyberPuppy Software, Inc.
cfh@cyberpuppy.com

Abstract

This paper provides an introduction to the Simple Mail Transport Protocol. It explains the basic state machine that describes the protocol, and details the flow of information that encompasses a mail transaction. A Metrowerks PowerPlant implementation of the basic SMTP mechanism is presented, with details on how to get started with the PowerPlant networking classes. Some of the issues you must watch out for with the current class framework are revealed.

Introduction

With all of the excitement over the last twelve months regarding the Internet, one aspect has been curiously absent. With the staggering growth of the Internet, both in size measured by interconnected computers and in people, you do not frequently hear what is occurring in the children's market space. New versions of Netscape Navigator, Microsoft Explorer, and others are wonderful tools, but they assume that the user will be an individual who can find her own way around the networks, without the need for coaching or protection.

Consider the experience you have while browsing various web sites. Although most computer-savvy individuals have no problem with the abstract concept of "cyberspace", most new computer users–including many younger children–have trouble pinning down network locations within a spatial frame of reference. This feeling is exasperated by the fact that while you may be browsing a site at the same time as many other users, you never get a sense of the community that forms around that site's content.

To some degree, the problem of community has been addressed over the years via chatting technologies (IRC, MUD/MOOs, etc.). Until recently, this has remained a separate domain from web and other information technologies.

My company decided to tackle the problem of community from a new angle; providing a space for children to gather, find one-another, and make new pen-pals. The technologies would emphasize interpersonal interaction while providing for seamless integration of static and dynamic content. From this goal, the product PigMail was born.

One feature of PigMail is the support for plugins that can provide new tools and toys to the user. This paper briefly discusses some of the information that was uncovered in creating a very simple, very small module for sending and receiving email. In particular, it discusses an implementation of the Simple Mail Transport Protocol (SMTP) [RFC821] using the Metrowerks PowerPlant network classes. Hopefully this treatment will provide a helpful starting point for individuals interested in working on such projects. The paper assumes a basic familiarity with PowerPlant and networking. A good introduction to networking in general, and TCP/IP development in particular is [COMER91] and [TANNEN81].

The functional requirements of the simple emailer include the ability to send email via SMTP as single shots. Therefore, there is no need to keep the message around. The user types a new message or replies to some other message, and when done, immediately initiates a connection, sends the message, and then throws away the document. When on the receiving end, mail messages are gathered one at a time from a mail host using the Post Office Protocol (POP3) [RFC1725], and stored locally within a simple mail container file. The target system includes MacOS 7.1 or newer, and either Open Transport or MacTCP. My discussion below uses CodeWarrior 8, PowerPlant, and some of the netbourne patches.

\
Issues such as performance and memory management are not included in this discussion, although memory usage is always a concern when trying to create small, tight modules. Also, implementation of a mail agent to extract messages from mailboxes is not discussed in this document. See [RFC1725] and others for discussions about mail retrieval protocols such as POP3 and IMAP4 [RFC1730].

The Simple Mail Transport Protocol (SMTP)

The Simple Mail Transport Protocol was designed to be an easily implemented, reliable mechanism for moving mail messages from one trusted host to another. This discussion provides an overview of the protocol, but the definitive specification is [RFC821]. [COMER91] provides an alternate treatment of this material.

SMTP as a protocol is specified independently of a transport service. This paper describes an SMTP implementation using TCP, which is the most common transport medium in use today for SMTP on microcomputers. SMTP is assigned to the permanent TCP port 25.

The SMTP specification describes a lock-step protocol in which the sender and the receiver transmit very specifically formatted messages to one another, awaiting a response before continuing. At a high level, the SMTP architecture can be described by a simple finite state machine which contains three main states, and one intermediate state. See Figure 1 below.

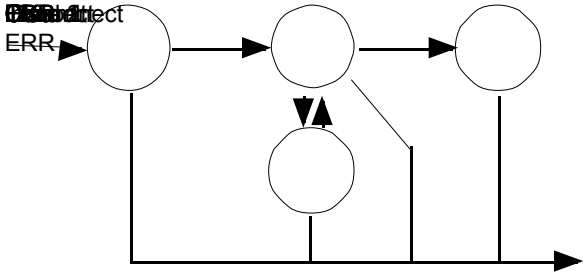

Figure 1. SMTP Finite State Machine

SMTP defines a small, required command set, with several optional commands included for convenience purposes. Table 1 shows the minimal set required for a SMTP sending client.

HELO	- Initial State Identification
MAIL	- Mail Sender Reverse Path
RCPT	- One Recipient's Forward Path
DATA	- Mail Message Text State
RSET	- Abort Transaction and Reset all buffers
NOOP	- No Operation
QUIT	- Commit Message and Close Channel

Table 1. Minimum SMTP Command Set

Commands may have zero or more parameters. Commands and their parameters are issued as ASCII plain text strings. A command is terminated with a carriage-return, line-feed (<CRLF>) pair. Commands do not span lines, the termination pair completes the command line. See example 1.

```
MAIL FROM:<cfh@cyberpuppy.com><CRLF>
```

Example 1. A Mail Sender command

Acknowledgment messages are formed by a three digit return code, followed by optional text. The three digits represent error and success codes. See example 2 below. Note that only the first three digits are significant within an acknowledgment message. The textual portion of the reply messages is for human understanding and can contain any text. Messages are grouped by meaning by using the first digit as a key. Messages beginning with a "2" are success messages, "3"'s are error codes, etc.

```
250 Requested mail action okay<CRLF>
```

Example 2. Typical Reply Acknowledgment

Normally, the acknowledging process will send one reply message per command. Each reply is ended with the standard <CRLF> token. It is possible that more than one acknowledgment message may be sent and this is not prohibited by the protocol specification. You should consider that some servers may generate more than one line of response and handle that case accordingly–this occurs most frequently with message 220, the service ready message transmitted on startup from the receiver when the sender initiates a connection. If you aren't careful, this can throw your state machine off. The

\

updated SMTP specification states that multiline responses should include a hyphen ("-") immediately following the result code of each intermediate status code. The final result line is formatted normally.

A typical SMTP session can be characterized as shown in Figure 2 and described here. The sender ([S]) opens a two-way channel to the receiver ([R]). The receiver can be the final destination or an intermediate node described in the messages path explicitly or implicitly by network routing tables. At connect time, both hosts are in the Initial state. [R] sends an acknowledgment that the channel is open. [S] sends a HELO message, identifying itself to the receiver. Note that authentication is not required, so it is very easy to spoof sender IP addresses. SMTP is not a secure messaging protocol. [R] sends back a success or error message, possibly denying access to the sender. If the HELO was successful, both sides are now in the Data 1 state. [S] sends a MAIL command describing the sending party's fully qualified reverse path. [R] acknowledges the successful receipt of the path and clears all of its transaction buffers. [S] sends one or more RCPT commands describing the forward path of recipients of the mail message, one recipient per line. [R] accepts or rejects each address.

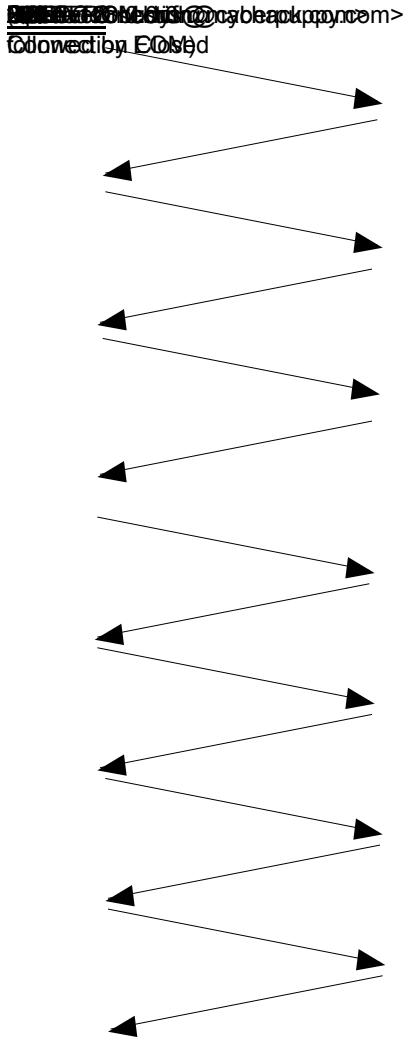

Figure 2. Typical SMTP Transaction Data Flow

[S] now sends a DATA command, instructing the receiver that all following data is the actual mail message, thereby putting the transaction in the Data 2 state. Transmission of the message text completes when the end-of-message (EOM) sequence is sent (a <CRLF>.<CRLF> triplet). Data transparency is achieved by stuffing any instance of the EOM sequence occurring within the body of a message with a period "." character prefix. The receiver checks each line for a leading period and removes it before buffering the data. Only when [R] detects the "real", tailing EOM, does it send an acknowledgment.

[S] sends a QUIT command to place the transaction in the Commit state. [R] acknowledges the

\
command and closes the channel. It then delivers the message to the recipients' mailboxes or forwards the message on to the next server in the recipients' forward paths.

You will note that the SMTP protocol does not handle any of the fields you would associate with a standard mail message (fields such as Subject:, Reply-To:, etc.). These fields, which make up a message that conforms to [RFC822], are built and parsed by the mail handling agent on either end of the SMTP transaction. SMTP treats the mail message in an opaque manner, sending the headers and message body all at once during the Data state. The SMTP code only peeks at the message to ascertain EOM transparency conditions.

Implementing SMTP in PowerPlant

To implement a simple SMTP client for the PigMail project, I chose to create a simple mail editor and tie it to the SMTP code by using a `LSingleDoc` derived class and its associated window member. Initially, this implementation used a threaded approach. Soon after, debugging of the PowerPlant network classes bogged things down. I muttered "Keep It Simple Stupid" to myself a couple of times and created the very simple, event-loop based asynchronous version which is presented here.

The threaded implementation is actually not much more difficult to construct, but it does obscure the discussion at this introductory level. However, because SMTP is a simple problem domain, it is a great opportunity for experimenting with threading. You could implement the entire SMTP state machine as one thread, to which you hand off all data and let it rip. Or, you could be creative and implement a two thread approach and play with the Producer/Consumer model of cooperative processes [SILBER92]. I tried both, and while they work fine, they violated my KISS requirement. The most important thing I learned with these experiments was the danger of mixing threads which operate with different PowerPlant drawing contexts, talk about major view foci problems!

The simple mail sender class displays a window which contains three text edit fields: destination address, subject, and message body. It also contains a control to send the message when done and a status field. Figure 3 shows a picture of the interface.

\

Figure 3. Simple Mail Editor

In the event-loop/asynchronous handling implementation below, I began by creating a `LSingleDoc`, `LAsyncClient` class similar to the one shown in Listing 1. [Note: To facilitate your understanding in the following walk through, you may want to refer to the sample source code that should be supplied with this paper's distribution.]

```
class CPMailEditorDoc : public LSingleDoc, public LListener, public LAsyncClient {
public:
                        CPMailEditorDoc(LCommander *inSuper,
                                    const LStr255 &inTo = "",
                                    const LStr255 &inSubj = "");
      virtual                       ~CPMailEditorDoc();

      virtual     void        ListenToMessage(MessageT inMessage, void *ioParam);
      virtual     void        Connect();
      virtual     void        Disconnect();
      virtual     Boolean     IsIdle();
      virtual     Boolean     AllowSubRemoval(LCommander* inSub);

protected:
      virtual     void        HandleAsyncMessage(const LAsyncMessage& inMessage);
                  void        BuildSessionWindow(void);
                  void        SendMailMessage(void);
                  void        RunMachine(char *inDataBuffer, Uint32 inDataSize);
                  void        SendHELO(Boolean inUseShort = false);
                  void        SendQUIT(void);
                  void        SendRSET(void);
                  void        SendNOOP(void);
                  void        SendMAIL(void);
                  void        SendRCPT(LStr255 &inRecipient);
```

```
\
                    Boolean     SendNext(void);
                    void        SendDATA(void);
                    void        SendBody(void);
                    void        SendHeader(void);
                    Boolean     ParseReply(char *inBuffer, Uint32 inBufferLen,
                                           Uint32& inPos);

      MailPreferenceTypeH     mMailPrefs;
      LStr255                 mTo;
      LStr255                 mSubject;
      Handle                  mBody;
      LEndpoint*              mEndpoint;
      LCaption*               mStatusPane;
      Int32                   mMachineState;
      Int32                   mLastCode;
      Int32                   mCurPos;
      Int32                   mMachineReplyState;
      char                    statusBuffer[8];
};
```

Listing 1. SMTPClient Class Definition.

The constructor initializes all member data, and calls the `::BuildSessionWindow()` member function to create the interface. The To: and Subject: fields are filled with optional data supplied by the caller of the constructor.

At this point, control rests within the standard PowerPlant event mechanism, and the user can interact with the editor. When her message is done, she presses the Send button, and away we go.

The `SMTPClient` class receives the button message via its `::ListenToMessage()` method. Here we call a `::SendMailMessage()` method. `::SendMailMessage()` extracts the data from the UI and initiates a connection.

The `SMTPClient ::Connect()` method makes use of a wonderful PowerPlant object called the `UNetworkFactory`. This object allows you to use the best transport mechanism installed at run time. It will automatically switch between Open Transport and "Classic Networking" (aka MacTCP) depending upon which is active at the time the `UNetworkFactory` is called. We create an asynchronous endpoint object that uses the event-loop to receive incoming asynch messages. An endpoint is simply an object that represents one-half of the communication link.

After creating the endpoint, we bind it to a network address. We specify both the address information for the originator–the sender's host–and the SMTP server host. Listing 2 shows the connection sequence.

```
// ---------------------------------------------------------------------------
//          • Connect
// ---------------------------------------------------------------------------
void CPMailEditorDoc::Connect()
{
      mEndpoint = UNetworkFactory::CreateTCPEndpoint(
                                    UNetworkFactory::Asynchronous(this));
      ThrowIfNil_(mEndpoint);

      // Initialization: Bind to any local port,
      // then connect to the remote host.
      LInternetIPAddress address(0, 0);
      mEndpoint->Bind(address);
```

\

```
	LInternetDNSAddress remoteAddress((**mMailPrefs).smtpHost, kSMTPPort);
	mEndpoint->Connect(remoteAddress);
}
```

Listing 2. The Connection Method

The asynchronous networking mechanism in PowerPlant is very easy to use. When network commands complete, or incoming messages are received, the networking classes call your `LAsyncClient` object's `::HandleAsyncMessage()` method. Here you can crack the incoming message and dispatch to your various handlers. Listing 3 shows the how simple the `::HandleAsyncMessage()` dispatch mechanism is.

When we are establishing the initial connection, as soon as we are notified that the connection is created, we set our endpoint to be in auto-receive mode. This endpoint mode automatically issues a receive command on your connection, thereby catching all data that is sent to your client without needing to explicitly issue receive commands.

In the `SMTPClient` code, whenever we get something from the SMTP server, we send that in to our SMTP state machine (the `::RunMachine()` method). `::RunMachine()` alternates between parsing incoming messages for their response codes and sending the next appropriate SMTP command.

```
// ---------------------------------------------------------------------------
//		• HandleAsyncMessage
// ---------------------------------------------------------------------------
void CPMailEditorDoc::HandleAsyncMessage(const LAsyncMessage& inMessage)
{
	switch (inMessage.GetMessageType()) {
		case T_DISCONNECT:
		case T_ORDREL:
			mEndpoint->AcceptDisconnect();
			// fall thru as the connection is closed at the other end
			// and we won't necessarily get the DISCONNECTCOMPLETE
			// when we issue an AcceptDisconnect instead of a Disconnect
		case T_DISCONNECTCOMPLETE:
			delete this;
			break;

		case T_CONNECT:
		case T_PASSCON:
			if (inMessage.GetResultCode() == noErr)
				mEndpoint->AutoReceive();
			break;

		case T_DATA:
		case T_EXDATA:
			LDataArrived* data = (LDataArrived*) &inMessage;
			if (data->GetDataSize())
				RunMachine((char *) data->GetDataBuffer(),
									data->GetDataSize());
			break;
	}
}
```

Listing 3. The HandleAsyncMessage Method.

\
Listing 4 shows part of the `::RunMachine()` method, which is an example implementation of the SMTP state machine described above. This implementation is a little unusual, and probably a little less clear, because its external switch statement jumps between result codes, while the inner conditionals branch on the actual state of the system. This code folds the alternating Reply/Send states together. Most of the time, the machine will be receiving state code 250 (success) and staying within the first case. Later status cases cover initialization, rundown, and error conditions.

```
// ---------------------------------------------------------------------------
//    • RunMachine
// ---------------------------------------------------------------------------
void CPMailEditorDoc::RunMachine(char *inDataBuffer, Uint32 inDataSize)
{
   Uint32   thePosition = 0;
   Boolean done = false;

   while (!done && thePosition < inDataSize)
      if (ParseReply(inDataBuffer, inDataSize, thePosition))
      {
         switch (mLastCode) {
            case 251:   // ok, but non-local user
            case 250:   // success
               switch (mMachineState) {
                  case eGreetingLong:
                  case eGreetingShort:
                     SendMAIL();
                     mMachineState = eMailSender;
                     break;

                  case eMailSender:
                     if (SendNext())
                        mMachineState = eMailDestination;
                     else
                        mMachineState = eMailInitiateData;
                     break;

                  case eMailDestination:
                     if (!SendNext())
                        mMachineState = eMailInitiateData;
                     break;

                  case eMailInitiateData:
                     SendDATA();
                     mMachineState = eMailBody;
                     break;

                  case eQuitting:
                     SendQUIT();
                     mMachineState = eDisconnecting;

                     break;

                  case eDisconnecting:
                     break;

                  default:
```

```
\
                    Assert_(false);
                    break;
                }
```

```
\
                break;
/*
...error cases and intermediate data case removed...see sample source
*/
          }
       }
}
```

Listing 4. SMTP State Machine Sample Implementation (Partial)

`::ParseReply()` collects the return information from the server and breaks out the result code. It discards the extra textual information. The `Send` methods simply format the corresponding commands with any parameters and push them out the endpoint. Listing 5 shows a typical Send method. Note that the `::SendNext()` method actually parses the To: field's data to allow for more than one destination address. In this way, the user can specify a comma delimited list of mail addresses. SMTP only allows one forward path per RCPT command, so we have to cycle through the *n* addresses sequentially.

```
// ---------------------------------------------------------------------------
//          • SendQUIT
// ---------------------------------------------------------------------------
void CPMailEditorDoc::SendQUIT(void)
{
      LStr255 param("QUIT");
      if (mStatusPane)
            mStatusPane->SetDescriptor((StringPtr) param);
      param += kCRLF;
      Int32 theSize = param.Length();
      mEndpoint->Send(&param[1], theSize);
}
```

Listing 5. Typical Command Send Method

When we get to the Data state, we begin by formatting and sending a [RFC822] header. The header minimally includes return path information, subject, recipient address information, and a properly formatted date field. Other fields are optionally appended to the header. The client then sends each line of the message body, testing each line for instances of the EOM sequence and properly byte-stuffs those lines.

At the end of the message body, an EOM is sent. Assuming that the server has accepted the transaction up until this point, we send a QUIT command, which commits this transaction.

The QUIT causes the SMTP server to send an acknowledgment and close the TCP channel from that end. The `LAsyncClient` object receives a T_ORDREL message requesting an orderly shutdown of the endpoint. The endpoint accepts the disconnect request and then deletes itself.

Assuming that no error messages were encountered, we just sent an Internet mail message via SMTP!

Implementation Problems

During this exercise, I encountered several gotchas with PowerPlant. Here I will try to explain them. Note that some of these issues are planned to be fixed within the PowerPlant release for CodeWarrior 9. The bugs and problems have been reported to Metrowerks.

Endpoints in their current implementation are tricky beasts. One problem with the asynchronous model is that you can get unusual dependencies that are not normally expected. One of the great current mysteries of the PowerPlant networking classes is when to properly destroy an endpoint. The

\

asynchronous messages which tell an `LAsyncClient` what is happening are allocated out of a pool of memory created by the `LEndpoint`'s Notifier object [most of the time, actually there is a "bug" in that some `LNetMessage` objects are created from the endpoint's pool is CW8]. The notifier is destroyed by the endpoint when the endpoint is destroyed. Unfortunately, if you delete your endpoint from within `::HandleAsyncMessage()` when you receive a message–such as T-DISCONNECTCOMPLETE–you will kill the memory pool from which the message is currently allocated. This causes problems when the message call stack pops back to the message sending method, and then tries to delete itself again. Boom!

Our destructor (see Listing 6) defers the deletion of the `LEndpoint` object. In this example, we spawn a thread to handle the deletion. This is clearly a work around, and an official solution may exist by CodeWarrior 9's time-frame.

```
// ---------------------------------------------------------------------------
//          • dtor
// ---------------------------------------------------------------------------
CPMailEditorDoc::~CPMailEditorDoc()
{
      // tell notifiers to bug out
      ClientIsClosing();

      if (mMailPrefs)
            ::DisposeHandle((Handle) mMailPrefs);

      // close endpoint if any
      if (mEndpoint)
      {
            // try to defer the deletion of the endpoint
            LSimpleThread *aThread =
                  NEW LSimpleThread((ThreadProc) DeleteEndpointObject,
                                    (void*) mEndpoint);
            if (aThread)
                  aThread->Resume();
            else
                  delete mEndpoint;
            mEndpoint = nil;
      }
}
```

Listing 6. Destructor Defers Endpoint Deletion

A second problem can occur due to overflow problems on sending data. The current endpoint implementations of the `Send` method do not notify you if the outgoing data has caused an overflow condition. This can happen if you are relying on the auto-send mechanism which copies your data to an intermediate buffer. If you are generating data to go out more quickly that it can be sent, or if you try to send chunks larger than the pool can accommodate, the send will fail silently. There is the beginning of a mechanism to implement a notification of this error, but it is not complete in the CodeWarrior 8 release. You will need to apply a work around in your own code and/or modify the class to patch this up. In my sample code, I simply return the size of the buffer sent or the result code by changing the `Send` methods to make the data size parameter a reference copy (e.g. `void LMacTCPEndpoint::Send( void* inData, Uint32& ioDataSize, LNotifier* inNotifier)`).

\
Another significant gotcha is encountered when you implement a scheme in which you take very small pieces of data out of the incoming data stream. By making repeated calls to the endpoint's `Receive` method with a buffer size of one–or other small sizes–you will quickly run out of pool space. Or so it will seem. On closer examination, you will note that there is sufficient space within the pool, but it is impossible to allocate space for your receive request. This fragmentation of the endpoint pools can be avoided by using all data immediately when you get a T-DATA or T-EXDATA message.

If you must receive data in extremely small pieces, you can implement a more sophisticated free block coalescing algorithm, or some kind of intermediate buffering scheme. Using all of the data immediately appears to be the most efficient mechanism at this time.

A final problem with the networking classes is a collection of memory leaks. There are some instances of exceptions being raised before deleting memory allocated from a pool. See `LMacTCPEndpoint::Receive()` for an example. These leaks are being cleaned up in the CodeWarrior 9 release.

Conclusion

This paper has provided an introduction to the Simple Mail Transport Protocol. It explained the basic state machine that describes the protocol, and details the flow of information that encompasses a mail transaction.

A Metrowerks PowerPlant implementation of the basic SMTP mechanism is presented, with details on how to get started with the PowerPlant networking classes. Some of the issues you must watch out for with the current class framework are revealed.

The goal of this paper has been to get you started on your own experiments with the networking classes in general, and SMTP in particular. With the provided implementation, you should be able to add mail sending via SMTP to your project in short order.

If you use any of this information or code and have suggestions, improvements, or questions, please let me know (and send a copy of your cool application!)

Bibliography

[COMER91] Comer, Douglas E.. *Internetworking with TCP/IP Volume I: Principles, Protocols, and Architecture*. Prentice Hall, Englewood Cliffs, New Jersey. 1991.

[RFC821] Crocker, D.H. "Standard For The Format of ARPA Internet Text Messages," ARPANET Request for Comments, No. 821. SRI International: Menlo Park. August 1982.

[RFC822] Postel, J.B. "Simple Mail Transfer Protocol," ARPANET Request for Comments, No. 822. SRI International: Menlo Park. August 1982.

[RFC1725] Myers, J.. "Post Office Protocol - Version 3," ARPANET Request for Comments, No. 1725. SRI International: Menlo Park. November 1994.

[RFC1730] Crispin, M.. "Internet Message Access Protocol - Version 4," ARPANET Request for Comments, No. 1730. University of Washington. . December 1994.

[SILBER91] Silberschatz, Abraham, Peterson, James C., et. al.. *Operating System Concepts, 3rd Edition.* Addison-Wesley Publishing Company, Reading, Mass.. 1991.

[TANNEN81] Tannenbaum, Andrew. *Computer Networks*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1981.