Programmer Guidelines for the **MAEstro** Multimedia Authoring Environment

George D. Drapeau
Stanford University
drapeau@sioux.stanford.edu

October 8, 1991

Contents

1 Introduction

The purpose of this document is to help programmers write applications for the **MAEstro** authoring environment. The document gives an overview of the environment, describes the tools available to the application programmer, and shows how to correctly use these tools.

The **MAEstro** project was created in an effort to allow simple, widespread access to new media in the computing environment. The focus of the **MAEstro** project is authorship — the environment should be simple and complete enough for any student or faculty member to be able to author his own multimedia documents.

The **MAEstro** environment consists of a suite of applications and an inter-application messaging system. There is one application for each medium in the authoring environment; In addition, there is one application responsible for authorship; it generally does not control media directly, but sends messages to the other applications in the environment via the network protocol, effectively controlling media by "remote control". The network protocol is inserted into each application, enabling all applications in the environment to communicate with each other.

2 System Overview

The **MAEstro** environment is a distributed, network-transparent, multimedia authoring environment. One of the basic assumptions underlying the design of **MAEstro** is that an authoring application (i.e., an application that facilitates the creation of multimedia documents) should not need any specific knowledge of any media included in the document. This assumption allows the authoring application to incorporate new media without rewriting the authoring application. To allow the authoring application complete flexibility with regard to the media included in its documents, a messaging system was designed to allow the authoring application to send messages to other applications in the environment. During authorship, the authoring application sends messages to the other applications asking them what they are doing. For example, the authoring application might ask the CD editor what selection of music it is playing so that section of music can be included in the multimedia document. During playback, the authoring application sends messages to the other applications telling them to replay the selections it earlier asked them about.

The authoring environment described above consists of the following components:

- **Media editors.** These are applications that directly control media. Each application is responsible for one medium; for example, the CD editor deals only with CD audio and does not concern itself with the control of videodiscs; likewise, the videodisc editor knows nothing of CD audio.

- **The authoring application.** This is the key application in the environment; it sends messages to the media editors to coordinate their actions and to play segments of those media at the appropriate times. It may or may not deal directly with any media; the environment is designed so that the authoring application can control any medium via "remote control", that is, by sending messages to the media editors, making the the media editors perform on behalf of the authoring application.

- **A network protocol.** This is a program library consisting of a set of functions that facilitate inter-application communication. Included are a standard set of messages to request that remote applications open a document, to ask the remote application for the name of its currently opened document, to select part of a document, to ask for the current selection within a document, and to tell the remote application to perform the document's current selection.

- **The Port Manager application.** This is a program that starts when the computer boots and remains running at all times. It serves as a central authority with which applications register themselves; by registering themselves, applications announce that they are ready to receive requests from other applications. The Port Manager helps applications make initial contact with each other.

  For more explanation of these components, see [Usenix].

3 The MAEstro Messaging System

The **MAEstro** Messaging System is a program library that facilitates communication among applications over the network. The protocol has a relatively small set of messages designed to support simple operations on documents, plus messages for establishing, confirming, and ending communications with other applications.

The two objects "Sender" and "Receiver" handle all of the message delivery in the **MAEstro** Protocol. They in turn make use of several key data structures, the most important of which is the Port.

The remainder of this section will describe the messages comprising the **MAEstro** Protocol, operation of the Sender object, operation of the Receiver object, and the components of the Port data structure.

3.1 MAEstro Messages

The **MAEstro** protocol consists of the following set of messages:

- **OpenDocument**
- **GetCurrentDocName**
- **GetSelection**
- **SetSelection**
- **PerformSelection**
- **ConnectWithPortMgr**
- **GetOpenApps**
- **GetPortFromName**
- **DisconnectFromPortMgr**
- **Ping**
- **HaltSelection**
- **PauseSelection**
- **ResumeSelection**
- **HideApplication**
- **ShowApplication**
- **GetAppIcon**

The Sender and Receiver objects have a corresponding method for each of these messages. For example, there is a `SenderOpenDocument()` method and a `ReceiverOpenDocument()` method. `SenderOpenDocument()` is used to send a request to another application to open a particular document; `ReceiverOpenDocument()` is used to fulfill an OpenDocument request sent from another application.

Any application that uses the Sender and Receiver messaging objects is eligible to send and receive the above messages. The messages are briefly described here; subsequent sections on Senders and Receivers will describe the arguments used by each method.

The following messages are sent from one application to another application:

- `OpenDocument`
  An application sends this message to ask another application to open a particular document.

- `GetCurrentDocName`
  An application sends this message to ask another application for the name of its currently open or "active" document.

- `GetSelection`
  An application sends this message to ask another application for information about its current selection. The format of this information, called a `Selection`, will be described later.

- `SetSelection`
  An application sends this message to ask another application to select part of a current document. The requesting application sends a Selection indicating what part of a document is to be selected.

- `PerformSelection`
  An application sends this message to ask another application to "perform" the current selection. For a text editor, this may mean something as simple as highlighting a region of text; for a CD player, it may mean to play a particular section of the CD.

- `HaltSelection`
  An application sends this message to ask another application to halt performance of its current selection. For a CD player, this may mean to stop the audio and park the playback head.

- `PauseSelection`
  An application sends this message to ask another application to pause performance of its current selection. The semantics are different than those of `HaltSelection`; the assumption made by a `PauseSelection` message is that the interrupted selection will soon be resumed. For a CD player, this may mean to pause the CD but leave the playback head in place so the audio can quickly be resumed.

- `ResumeSelection`
  An application sends this message to ask another application to resume performance of its current selection. This assumes that there is a current selection and that performance of that selection has just been interrupted (via `PauseSelection`).

- `HideApplication`
  An application sends this mesasge to ask another application to hide itself (i.e., to remove its windows from the screen, most likely by "iconifying" itself). An authoring application might send this message to clear screen space taken by several applications.

- `ShowApplication`
  An application sends this message to ask another application to show itself (for example, if the application were "iconified"). An authoring application might send this message to bring a particular media editor to the front of the window stack, allowing the author to work with that editor.

- `GetAppIcon`
  An application sends this message to ask another application for an icon that should uniquely identify it. An authoring application might ask for the media editors' icons so that it could present a palette of those icons to the author, showing the author which media are currently available for editing.

The following messages are sent to the Port Manager:

- `ConnectWithPortMgr`
  This function initiates a connection with the Port Manager, telling the Port Manager about the application's name, port number, and hostname. By doing this, an application tells the world that it is ready to receive messages from other applications. An application usually does not explicitly call this function; it is usually called by the `NewReceiver()` method.

- `GetOpenApps`
  This function asks the Port Manager to return a list of the applications currently advertising their services. This is how applications find out about the existence of other applications.

- `GetPortFromName`
  An application sends this message to the Port Manager when the application wants to communicate with a particular application. The Port Manager will return the Port information for the requested application if possible. If necessary, the requested application will be launched.

- `DisconnectFromPortMgr`
  An application sends this message to the Port Manager when it is no longer willing to listen for network messages. Usually this is called just before the application quits.

The remaining message is used primarily by the Port Manager, but any application can use it:

- `Ping`
  This message is used to check if the receiving application is still listening for messages.

3.2 The Sender Object

A Sender is an object used to send messages to other applications. You create one Sender for each application with which you will communicate. Since every application registers itself with the PortManager, every application has at least one Sender (the one that's responsible for communicating with the PortManager).

To create a Sender through which to send messages, use the `NewSender()` method. `NewSender()` takes a pointer to a `Port` as its only argument; the `Port` tells the Sender where the remote application is (see the section on 3.8).

Here's an example showing how an application would create a connection with the Port Manager:

```
Sender*        sender;
Port           senderPort;

senderPort.hostName = ``localhost'';
senderPort.portNumber = PortMgrPortNumber;
sender = NewSender(&senderPort);
```

The Port `senderPort` is filled in with information about what host and port number to connect with. The defined constant `PortMgrPortNumber` is provided by the library. After the call to `NewSender()`, the variable `sender` should be checked for a NULL value. If `sender` is NULL (i.e., if `sender == (Sender*)NULL`), then the NewSender method was unable to establish a connection with the remote application (in this case, the Port Manager). If the Sender returned by `NewSender()` is non-NULL, then a connection to the remote application was successfully established.

Now that a new Sender has been created, the application can now send messages to the PortManager. If your application needs to send messages to other applications, it must create a new Sender for each application with which it will communicate.

When your application is finished with a Sender, call `DestroySender()`. This method will close the network connection to the remote application and free the space taken by the Sender object.

3.3 Sender Methods

Every Sender method (with the exception of `NewSender*()` and `DestroySender()`) returns an `int`. A return value of zero always means that the method completed successfully (i.e., that the network message was successfully delivered).

Each Sender method (with the exception of `NewSender()` takes a `Sender*` as its first argument. The `Sender*` points to a Sender containing information about the remote application's network address. In other words, each Sender method needs information about the application to which the message should be sent.

- `int SenderOpenDocument(Sender* sender, char* documentName)`
  This method is used to ask another application to open the document named in the string passed in as argument. For example, to ask the "TextEditor" application to open the document "/tmp/letterToMom", your application would do the following:

  ```
  result = SenderOpenDocument(textEditSender,''/tmp/letterToMom'');
  ```

- `int SenderGetCurrentDocName(Sender* sender, char** documentNameReturn)`
  This method is used to ask another application for the name of its currently open document. The space needed to hold `documentNameReturn` is allocated for you; your application should free the space taken by the string when it is no longer needed.

- `int SenderSetSelection(Sender* sender, Selection* selection)`
  This method is used to ask another application to select part of its currently open document. The semantics of this method are explained in section 8.2. Your application passes in a pointer to a Selection structure; your application must have already initialized the space for that Selection.

  The `SenderSetSelection()` method can be used to set a complete selection or part of a selection. Two fields in the `Selection` structure, the `duration` and `offset` fields, determine the semantics of the `SetSelection()` message. To understand the semantics of the message, consider a selection of duration 1000 milliseconds. The following cases cover the semantics possible with `SenderSetSelection()`:

  1. Offset is 0, duration is 1000. This defines a complete `Selection`, the default semantics most often used by the protocol.
  2. Offset is 500, duration > 500. An application sending these as `Selection` parameters is asking the remote application to play part of its current selection, beginning 500 milliseconds from the beginning of the selection and ending at the default end point.
  3. Offset is -500, duration is 1500. This means to perform more than the original selection, beginning 500 milliseconds before the original selection and ending at the default end point.
  4. Offset is 0, duration is 1500. This means to perform more than the original selection, beginning at the default start point and extending 500 milliseconds beyond the default end point.

  These cases show that the combination of the two fields `offset` and `duration` can specify partial selections, complete selections, and selections of greater length than originally intended. It is up to the receiving application to decide how to interpret these parameters, but the receiving application should do its best to satisfy the request sent by the calling application.

- `int SenderGetSelection(Sender* sender, Selection** selectionReturn)`
  This method is used to ask another application to return information about its current selection within its currently active document. The semantics of this method are explained in section 8.2. The space needed to hold the Selection in `selectionReturn` is allocated for you; your application should free the space taken by the Selection when it is no longer needed. Here's a code fragment showing how to use `SenderGetSelection()`:

  ```
  Selection*      theSelection = (Selection*)NULL;
  int             result = 0;

  result = SenderGetSelection(anAppSender, &theSelection);
  printf(''Selection duration is %d\n.'',theSelection->duration);
  ```

Remember to pass the address of the `Selection*` to `SenderGetSelection()`; if the function completes successfully, `theSelection` will point to a newly-allocated Selection structure that your application may use in whatever manner necessary.

- `int SenderPerformSelection(Sender* sender)`
  This method is used to ask another application to "perform" its current selection. The semantics of this method are explained in section 8.2.

- `int SenderConnectWithPortMgr(Sender* sender, Port* receivingPort)`
  This method is described above, in section 3.1. This method is not usually called directly from within an application; it is usually called by the `NewReceiver()` method. This method tells the receiving application (usually the PortManager) about the Port on which the calling application is listening for messages. The calling application fills in the receivingPort field before calling this function.

- `int SenderGetOpenApps(Sender* sender, PortArray** openAppsReturn)`
  This function is used to ask another application (usually the Port Manager) which applications are currently listening for messages. The space needed to hold the list of Ports returned in `openAppsReturn` is allocated for you; your application should free this space (using the `DestroyPortArray()` method) when no longer needed. See the section on 3.9 for more information on the `DestroyPortArray()` method.

- `int SenderGetPortFromName(Sender* sender,`
  `Port* appNameAndHost,`
  `PortArray** matchingPortsReturn)`
  This method is used to ask another application (usually the Port Manager) to fill in Port information about a specific application.

  The calling application asks for an remote application's address by setting the `appNameAndHost->appName` field to the name of the remote application being sought, and setting the `appNameAndHost->hostName` field to the defined constant `AnyHost`. If the host on which the remote application runs is also important (usually this is not the case), the calling application should fill in the `hostName` field with the name of the host on which the remote application should be running.

  For example, if an authoring application needed to communicate with "cdEdit" but the host was not relevant, the calling application would fill in the `appNameAndHost` argument as follows:

  ```
  appNameAndHost->appName = ``cdEdit'';
  appNameAndHost->hostName = AnyHost;
  ```

  If the calling application needed to communicate with the "cdEdit" running on the host named "crow", the calling application would fill in the `appNameAndHost` argument as follows:

  ```
  appNameAndHost->appName = ``cdEdit'';
  appNameAndHost->hostName = ``crow'';
  ```

  The calling application should allocate space for the `appNameAndHost` argument. If the application being sought is currently registered with the Port Manager, the `matchingPortsReturn` field will be filled with a list of Ports that match the information being requested. This space will be allocated for you; your application should free this space using the `DestroyPortArray()` method when done. See the section on 3.9 for more information on the `DestroyPortArray()` method.

  Further semantics of the `SenderGetPortFromName()` method are described in section 3.10.

- `int SenderPortNumber(Sender* sender)`
  This convenience function is used to get the port number of the application to which the `Sender` passed in as argument is connected. For example, if the `Sender` created by a call to `NewSender()` was connected to an application listening on port 3648, this method would return 3648.

- `int SenderDisconnectFromPortMgr(Sender* sender, Port* appPort)`
  This method is used to tell the Port Manager that the application is no longer listening for messages from other applications. This method is not usually called directly by the application programmer; it is usually called by the `DestroyReceiver()` method.

- `int SenderPing(Sender* sender)`
  This method is used to ask another application if it is still listening for messages from other applications. If the remote application is still listening, this method will return `0`; if the remote application is no longer listening for some reason, this method will return `-1`. If this method returns `-1`, the `Sender` passed in as argument should be destroyed, as the network connection used by the `Sender` is no longer valid.

- `SenderHaltSelection(Sender* sender)`
  This method is used to ask another application to halt performance of its current selection.

- `SenderPauseSelection(Sender* sender)`
  This method is used to ask another application to pause performance of its current selection.

- `SenderResumeSelection(Sender* sender)`
  This method is used to ask another application to resume performance of its current selection.

- `SenderHideApplication(Sender* sender)`
  This method is used to ask another application to hide itself.

- `SenderShowApplication(Sender* sender)`
  This method is used to ask another application to show itself.

- `SenderGetAppIcon(Sender* sender, IconData** iconDataReturn)`
  This method is used to ask another application for its application icon. The space needed to hold the IconData in `iconDataReturn` is allocated for you. Your application should free the space taken by the IconData when it is no longer needed.

3.4 The Receiver Object

A Receiver is an object that handles incoming messages from other applications. When a message arrives, the Receiver determines which message it was and calls the appropriate function. You can tell the Receiver to call your own function for each type of message it recognizes. If you don't specify your own functions, the Receiver will call default routines.

Applications need create only one Receiver; all incoming messages will be handled by that Receiver. However, your application may create more than one Receiver if there is some reason that the application should be listening for messages on more than one port.

3.4.1 Creating A New Receiver

To create a Receiver with which to listen for incoming messages, use the `NewReceiver()` method. By creating a new Receiver, an application advertises its services `NewReceiver` does this advertising by registering itself with the Port Manager; the information sent to the Port Manager will be discussed in section 3.8. In order for `NewReceiver` to register itself with the Port Manager, `NewReceiver` first needs a `Sender` with which to communicate to the Port Manager. Therefore, before creating a new Receiver, your application must first create a `Sender` setup to communicate with the Port Manager.

NewReceiver() is declared as follows:

```
Receiver*  NewReceiver(Sender*  portMgrSender,
                       char*    appName,
                       int      receivingPort)
```

The first argument is the `Sender` connected to the Port Manager with which the `Receiver` will register itself. The second argument is the name with which the `Receiver` will register itself. The third argument is the port number on which the `Receiver` is listening for messages. This argument is usually specified with the defined constant `AnyPort`, meaning that the `NewReceiver` method will assign a port number to the `Receiver` for you.

`NewReceiver()` will allocate the space for a `Receiver` and return a pointer to the new `Receiver`.

Here's an example showing how an application would advertise that it is listening for messages. For this example, assume that the `sender` was created as in the example from section 3.2. In this example, the name of the program under which the `Receiver` registers is "TestProgram".

```
Receiver*     receiver;

receiver = NewReceiver(sender,``TestProgram'',AnyPort);
if (receiver == (Receiver*) NULL)
  exit(1);
```

The `NewReceiver()` method uses the method `SenderConnectWithPortMgr()` to register itself with the Port Manager. One result of this is that the application will register itself with only one Port Manager. If your application will be registering with more than one Port Manager (for example, if your application provides a service that you'd like to offer across a large number of computers), your application may explicitly call `SenderConnectWithPortMgr()` after creating a new `Receiver`. Remember, to register with a different Port Manager, your application must first create a new `Sender` to communicate with that Port Manager. So if your application will be registering with ten Port Managers, your application will need to create ten `Sender's`.

3.4.2 Listening For Incoming Messages

Now that a new `Receiver` has been created, the application can now begin receiving incoming messages. For now this discussion will assume that it is not necessary to override the default `Receiver` message handling methods; the section 3.5 will discuss this.

There are several ways to begin listening for incoming messages:

- Some toolkits have built-in support for Sun RPC's (the message delivery system upon which **MAEstro** is written). Sun's XView toolkit is an example. To begin listening for incoming messages from other applications in XView, put the following line of code in your application:

  ```
  (void) notify_enable_rpc_svc (TRUE);
  ```

- If your application is doing nothing but listening for messages (for example, if your application does not have to deal with graphics or input from users, etc. — this is the traditional "server" model of computing), you can simply call the Sun RPC function `svc_run()`. This function never returns, so it should be the last line in your program's "main()" function.

- If your application deals with a window system and is using some sort of programmer's toolkit, there is probably some provision in the toolkit for calling a function at regular intervals. The NeXTStep, Motif, and XView toolkits all provide this functionality. For toolkits that do not have direct support for Sun RPC's but can call work procedures at regular intervals, the `ReceiverListenForMessages()` method is provided. `ReceiverListenForMessages()` takes no arguments; it scans its network ports for incoming messages and calls the appropriate message handlers if necessary.

  Your application should try to call `ReceiverListenForMessages()` at least once per second. The application may call the method as often as it wishes, but more than ten times per second is probably too often; the overhead of the method alone will probably begin to have a noticeable impact on the application's performance.

3.4.3 Destroying A Receiver

When your application will no longer be listening for incoming messages, call `DestroyReceiver()`. This method will send a message to the Port Manager requesting that its information be removed from the list of active applications. The method will then close down the incoming network port, then free the space taken by the `Receiver`.

The `DestroyReceiver()` method calls the `Sender` method `SenderDisconnectFromPortMgr()`. If your application has registered itself with more than one Port Manager, your application should explicitly call `SenderDisconnectFromPortMgr()` for every Port Manager with which it has registered except for one. Then the application should call `DestroyReceiver()` for the last Port Manager connection.

`DestroyReceiver()` is declared as follows:

```
void DestroyReceiver(Sender* portMgrSender, Receiver* receiver)
```

The first argument is the connection to the Port Manager with which the application is disconnecting. The second argument is the `Receiver` being destroyed.

The remaining Receiver object methods will be discussed in section 3.7.

3.5 Dispatch Tables

As provided by the **MAEstro** library, the `Receiver` methods don't do any useful work. The `Receiver` methods were meant to be overridden by the programmer so that the application can provide its own services and make them available to the network. Further discussion of which messages to override and for what purpose is in section 8.

To use your own methods instead of those provided by the library, use the data type `DispatchTable` and the method `BuildDispatchTable()`. The data type `DispatchTable` is a structure of pointers to functions, one for each method provided by the `Receiver` that has to do with processing incoming messages from other applications. `DispatchTable` is declared as follows:

```
typedef struct _DispatchTable
{
void            (*openDocumentPtr)(char**);
char**          (*getCurrentDocNamePtr)(void*);
Selection*      (*getSelectionPtr)(void*);
void            (*setSelectionPtr)(Selection*);
void            (*performSelectionPtr)(void*);
void            (*connectWithPortMgrPtr)(Port*);
PortArray*      (*getOpenAppsPtr)(void*);
PortArray*      (*getPortFromNamePtr)(Port*);
void            (*dispatchMessagePtr)(struct svc_req*, SVCXPRT*);
void            (*disconnectFromPortMgrPtr)(Port*);
void            (*pingPtr)(void*);
void            (*haltSelectionPtr)(void*);
void            (*pauseSelectionPtr)(void*);
void            (*resumeSelectionPtr)(void*);
void            (*hideApplicationPtr)(void*);
void            (*showApplicationPtr)(void*);
IconData*       (*getAppIconPtr)(void*);
} DispatchTable;
```

To override any of these methods with your own, create a variable of type `DispatchTable` and assign individual fields of that variable to the individual functions you have written. After filling in your own `DispatchTable`, call the method `BuildDispatchTable()` to use your functions.

Here's an example showing how your application would override the default `Receiver` methods to use your own. For the following code fragment, this discussion assumes that the functions `OpenDoc(), GetDoc(),`

GetSelection(), SetSelection(), and PerformSelection() have been declared and written elsewhere in your application. The fields that are filled in as NULL are services not provided by your application.

```
  static DispatchTable  dispatchTable =
  {
    OpenDoc,
    GetDoc,
    GetSelection,
    SetSelection,
    PerformSelection,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    HaltSelection,
    PauseSelection,
    ResumeSelection,
    HideApplication,
    ShowApplication,
    GetAppIcon,
    };

  BuildDispatchTable (&dispatchTable);
```

Note that the first field in the DispatchTable variable above is a pointer to a function called "OpenDoc"; it corresponds with the declaration of DispatchTable.openDocumentPtr above. Likewise, the "GetDoc" function overrides the default method ReceiverGetCurrentDocName(), and so on.

3.6 Using Your Own Message Handling Routines

If your application is to override any of the default Receiver methods, your application's functions must be declared in the same manner as the default Receiver methods (the declarations are discussed next) and the pointers to your application's function must be filled in the correct fields of the DispatchTable structure. For example, if your application is going to provide its own OpenDocument function, the pointer to that function must be filled in the first field of the DispatchTable structure, as in the example above.

Many of the Receiver methods have void as their return type; however, three of the methods available to be overridden return pointers to information. For example, the GetOpenDoc() method returns a pointer to a string.

> **Note: When overriding methods that return pointers to information, your functions must declare those return values as static.**

For example, if your application overrides the GetOpenDoc() method, it would be written in this way:

```
char** GetDoc()
{
  static char*  returnValue;

  returnValue = GetMyCurrentDocumentName();
  return(&returnValue);
}
```

The important part of this example is the declaration of the variable "returnValue" as a `static` string. This limitation is imposed by the Sun RPC messaging system and must be strictly followed.

3.7 Receiver Methods

Here are the declarations of all the `Receiver` methods:

- `Receiver* NewReceiver(Sender* portMgrSender,`
  `char* appName,`
  `int receivingPort)`
  This method registers an application with a Port Manager. The Port Manager with which to register is specified by the `portMgrSender` passed in as the first argument. The `portMgrSender` must therefore first be created by calling `NewSender()`; see section 3.2 for details.

  `NewReceiver()` allocates space for a `Receiver` and returns a pointer to the new `Receiver`. The second argument is a string indicating the name with which the application will register itself with the Port Manager. The third argument is the port number on which the `Receiver` will listen for messages. To make `NewReceiver()` allocate a port number for you, use the defined constant `AnyPort` as the third argument.

  If the method fails for some reason, it will return `(Receiver*)NULL`.

- `void DestroyReceiver(Sender* portMgrSender, Receiver* receiver)`
  This method is used to "unregister" an application with a Port Manager; that is, to tell a Port Manager that the `Receiver` is no longer listening for incoming messages. The Port Manager with which the application was registered is specified by the `portMgrSender` passed in as the first argument. The `portMgrSender` must have already been created by calling `NewSender()`; see section 3.2 for details.

  `DestroyReceiver()` frees the space pointed to by the method's second argument and closes the Sun RPC message service associated with that `Receiver`.

- `void ReceiverListenForMessages()`
  This method is used to check for any incoming network messages from other applications. If this method detects such a message, it will call the appropriate message handling routine. If your application has registered its own message handling routines as described in Section 3.5, `ReceiverListenForMessages()` will call your application's message handling routines when appropriate.

  This method is a convenience function to be used by toolkits that do not have built-in support for Sun RPC's. If your toolkit has support for Sun RPC's, there is no need to use this method.

  `ReceiverListenForMessages()` will return after processing one incoming message; it will also return if no messages are pending. Therefore, if your application is using this method to handle incoming messages, your application should repeatedly call this function, for example as part of a while loop that performs other tasks.

- `int ReceiverGetReceiverSocket(Receiver* receiver)`
  This is a convenience function, returning the socket through which incoming messages are being received.

- `void ReceiverOpenDocument(char** docName)`
  This method is called when a remote application sends a `SenderOpenDocument()` message. The semantics associated with this message usually involve the receiver of the message opening the document specified by the argument, but any semantics could be imposed. See section 8.2 for further discussion of the semantics of this message.

  The method takes one argument, a pointer to a string. To gain access to the string being sent, de-reference the pointer (e.g., `char* theDocToOpen = *docName`). The space taken by this string will be freed after this method exits, so if your application needs to keep track of this string, your application should make a copy of the string before exiting the method.

- `char** ReceiverGetCurrentDocName(void* unusedArg)`
  This method is called when a remote application sends a `SenderGetCurrentDocName()` message, asking your application for its currently active document. The semantics associated with this message usually involve the receiver of this message returning the filename of the application's currently open document (e.g., the name of a text file currently being edited in a word processor), but any semantics could be imposed. See section 8.2 for further discussion of the semantics of this message.

  The one argument to this method is a "dummy" argument necessary for the Sun RPC library, and should not be referenced. This method should return a pointer to a `static char*`:

```
static char* stringPointer = ``Untitled'';
return(&stringPointer);
```

  This space will not be freed, so your application should either recycle the space or free the space and re-allocate space the next time this method is called. For example, your `GetCurrentDocName()` method might look like this:

```
char** GetCurrentDocName(void* unusedArg)
{
  static char* theString = (char*)NULL;

  if (theString != (char*)NULL)
  {
    free(theString);
    theString = malloc(NumberOfBytesNeededHere);
  }
.
.
.
  return(&theString);
}
```

- `Selection* ReceiverGetSelection(void* unusedArg)`
  This method is called when a remote application sends a `SenderGetSelection()` message, asking your application for its currently active selection within its currently active document. The semantics associated with this message usually mean that the application should return information about the part of a document that the user has selected (e.g., the starting and ending byte numbers of selected text within a word processing document), but any semantics could be imposed. See section 8.2 for further discussion of the semantics of this message.

  The one argument to this method is a "dummy" argument necessary for the Sun RPC library, and should not be referenced. This method should return a pointer to a `static Selection`:

```
static Selection theSelection;
return(&theSelection);
```

  This space will not be freed, so your application should either recycle the space or free the space and re-allocate space the next time this method is called.

- `void ReceiverSetSelection(Selection* selection)`
  This method is called when a remote application sends a `SenderSetSelection()` message, asking your application to select part of the currently active document. The selection information is passed as the method's only argument. The semantics associated with this message usually mean that the application should use the

information passed in the `selection` argument to select part of the currently opened document (e.g., using the `start` and `end` fields as the starting and ending byte numbers of a word processing document), but any semantics could be imposed. See section 8.2 for further discussion of the semantics of this message.

The space taken by the `selection` argument will be freed after the method exits, so if your application needs to keep track of the `Selection` information, the application should make a copy of the `Selection` before exiting the method.

- `void ReceiverPerformSelection(void* unusedArg)`
  This method is called when a remote application sends a `SenderPerformSelection()` message, asking your application to "perform" the current selection within the currently active document. The interpretation of what "performance" means is left to the developer of each application. For example, performing a selection in a video editing application might mean to play a segment of video.

  The one argument to this method is a "dummy" argument necessary for the Sun RPC library, and should not be referenced.

- `void ReceiverConnectWithPortMgr(Port* app)`
  This method is called when a remote application sends a `SenderConnectWithPortMgr()` message. This message is usually sent by an application only to a Port Manager; at present, the only application that does anything useful upon receiving this message is the Port Manager. However, it is possible for your application to provide its own handling routine for this message.

  The `app` argument passed to this method contains information about the application advertising its services. The space taken by this argument will be freed after the method exits, so if your application needs to keep track of the `Port` information, the application should make a copy of the `Port` before exiting the method.

- `PortArray* ReceiverGetOpenApps(void* unusedArg)`
  This method is called when a remote application sends a `SenderGetOpenApps()` message. This message is usually sent by an application only to a Port Manager; at present, the only application that does anything useful upon receiving this message is the Port Manager. However, it is possible for your application to provide its own handling routine for this message.

  The one argument to this method is a "dummy" argument necessary for the Sun RPC library, and should not be referenced.

  This method should return a pointer to a `static PortArray`:

```
static PortArray openAppsList;
return(&openAppsList);
```

  The space pointed to by the `PortArray`'s `portArray` field will not be freed, so your application should either recycle the space or free the space (using `DestroyPortArray()`; see 3.9 for details) and re-allocate space the next time this method is called. For example, your `ReceiverGetOpenApps()` method might look like this:

```
PortArray* GetOpenApps(void* unusedArg)
{
  static PortArray listOfOpenApps = {(Port*)NULL, 0};

  if (listOfOpenApps.portArray != (Port*)NULL)
    {
      DestroyPortArray(&listOfOpenApps.portArray);
      listOfOpenApps.numberOfPorts = 0;
    }
```

```
...
...
...
  return(&listOfOpenApps);
}
```

- `PortArray* ReceiverGetPortFromName(Port* appPort)`
  This method is called when a remote application sends a `SenderGetPortFromName()` message. This message is usually sent by an application only to a Port Manager; at present, the only application that does anything useful upon receiving this message is the Port Manager. However, it is possible for your application to provide its own handling routine for this message.

  As performed by the Port Manager, this method returns a list of Port's in a `PortArray*` structure. This list represents the list of currently open applications that match the specifications passed in as the method's only argument. See the explanation of `SenderGetPortFromName()` for more information on the semantics of this message.

- `void ReceiverDispatchMessage(struct svc_req* requestPtr, SVCXPRT* transport)`
  This method is responsible for parsing incoming messages and calling the appropriate message handling routines. Your application will likely never have to override this method.

  The two arguments to this method are both Sun RPC arguments. The first argument is a pointer to information about the incoming message. The second argument is a pointer to information used by the `Receiver` object as a communication channel to and from the calling application.

  For more information about writing your own dispatching mechanism, refer to [SunRPC].

- `void ReceiverDisconnectFromPortMgr(Port* appPort)`
  This message is called when a remote application sends a `SenderDisconnectFromPortMgr()` message. This message is usually sent by an application only to a Port Manager; at present, the only application that does anything useful upon receiving this message is the Port Manager. However, it is possible for your application to provide its own handling routine for this message.

  The `appPort` passed in as argument contains information about the application that will no longer offer its services. The space taken by this argument will be freed after the method exits, so if your application needs to keep track of the `Port` information, the application should make a copy of the `Port` before exiting the method.

- `void ReceiverPing(void* unusedArg)`
  This method is called when a remote application sends a `SenderPing()` message, asking your application if it is still listening for messages.

  Since the purpose of this method is simply to check if an application is still alive, no data is passed between the sending and receiving applications. Your application will not likely need to override this method, but if it does, the method should spend as little time as possible to assure a quick reply to the calling application.

  The one argument to this method is a "dummy" argument necessary for the Sun RPC library, and should not be referenced.

- `void ReceiverHaltSelection(void* unusedArg`
  This method is called when a remote application sends a `SenderHaltSelection()` message, asking your application to halt performance of its current selection. If your application is not currently performing a selection, it should ignore this message.

- `void ReceiverPauseSelection(void* unusedArg)`
  This method is called when a remote application sends a `SenderPauseSelection()` message, asking your application to pause the performance of its current selection. If your application is currently performing a

selection, it should stop the performance immediately in such a way that performance could quickly resume. For example, a videodisc player would enter still-frame mode, so that the read head would be immediately ready to pick up where it left off.

If your application is not currently performing a selection, it should ignore this message.

- `void ReceiverResumeSelection(void* unusedArg)`
  This method is called when a remote application sends a `SenderResumeSelection()` message, asking your application to resume performance of its current selection. If your application is currently performing a selection and performance is paused, your application should resume performance as quickly as possible from the place the performance left off; it should not resume from the beginning of the selection.

  If your application is currently performing a selection but the performance is not paused, your application should ignore this message. Likewise, if your application is not currently performing a selection, your application should ignore this message.

- `void ReceiverHideApplication(void* unusedArg)`
  This method is called when a remote application sends a `SenderHideApplication()` message, asking your application to hide itself. The recommended action to take upon receipt of this message is for your application to "iconify" itself.

- `void ReceiverShowApplication(void* unusedArg)`
  This method is called when a remote application sends a `SenderShowApplication()` message, asking your application to show itself. If your application is currently "iconified", it should now do whatever it needs to do to "de-iconify" itself. If your application is not iconified but perhaps occluded by other application windows, your application should raise its windows to the front of the window stack, so that no other windows occlude your application's windows.

- `IconData* ReceiverGetAppIcon(void* unusedArg)`
  This method is called when a remote application sends a `SenderGetAppIcon()` message, asking your application for its application icon. All applications in the **MAEstro** environment should have an icon to represent themselves.

  This method should return a pointer to a `static IconData`:

  ```
  static IconData appIcon;
  return(&appIcon);
  ```

  This space will not be freed, so your application should either recycle the space or free the space and re-allocate space the next time this method is called.

- `void BuildDispatchTable(DispatchTable* table)`
  This method replaces a `Receiver`'s default set of message handling routines with those specified in the `table` passed in as argument. Only those fields that have been filled in as non-NULL will override the default handling routines; if you want your application to override one of the default routines with a routine that does nothing, filling in NULL as one of the fields in the `table` will not do the trick.

  For example, the default action when your application gets a `Ping` message is to print a statement to `stdout` saying that the Ping function was called. If you not want your application to print this message, you should override the default `ReceiverPing()` with one like this:

  ```
  void MyPingProc(void* unusedArg)
  {
   return;
  }
  ```

When you are filling in your own DispatchTable, fill in the pingPtr field to point to this routine, like this:

```
myDispatchTable.pingPtr = MyPingProc;
```

3.8 Ports

A Port is the address used to open a communication link with another application. The Port structure is defined as follows:

```
struct Port
{
  char* hostName;
  char* appName;
  int   portNumber;
};
```

The hostName field contains the name of the computer on which the application is running. The portNumber field contains the specific port number on which the application is listening. The appName field contains the name the application uses to register itself with the PortManager.

When your application advertises its services with the Port Manager, your application registers its services under a particular application name. For example, a video editing application might register itself as "VideoEdit". In this case, your application will create a new Port and assign the string ``VideoEdit'' to the appName field.

When your application registers itself with the Port Manager, the application usually only need fill in the appName field of a Port; the hostName and portNumber field are usually filled in automatically by the Sender and Receiver methods. If for some reason your application must explicitly fill in the other two fields, your application is allowed to do so. If your application is not going to fill in the hostName field, however, it should set the field to (char*)NULL.

3.9 The PortArray Structure

3.10 Communicating With Other Applications

To send messages to other applications, your application must first create a new Sender through which to send messages to the remote application. To create a new Sender, your application must pass in a Port describing the address of the remote application.

If your application will be communicating with a specific remote application, it can ask the Port Manager for the address of the remote application by using the SenderGetPortFromName() method. Your application must create a Port and fill in at least the appName field, and possibly also the hostName field. If your application needs to communicate with a specific application but the host on which that application is running doesn't matter, your application should only fill in the appName field and should set the hostName field to the defined string constant AnyHost. However, if your application needs to communicate with a specific application on a specific host (for example, if your application must communicate with a database on a specific machine when a local copy won't do), the application should also fill in the hostName field, specifying where the remote application is running.

When your application fills in the Port information and sends the SenderGetPortFromName() message to the Port Manager, the Port Manager will check its list of currently registered applications for all possible matches. The more specific the information your application passes in, the more specific the Port Manager will be in searching for matches.

For example, consider a Port Manager that has two applications currently registered as follows:

```
Application 1:
appName:        TextDataBase
hostName:       sioux.stanford.edu
portNumber:     2525
```

```
Application 2:
appName:        TextDataBase
hostName:       crow.stanford.edu
portNumber:     3042
```

If your application had filled in the `Port` information in the following way:

```
Port    myPort;

myPort.appName = ``TextDataBase'';
myPort.hostName = (char*)NULL;
```

the Port Manager would return a `PortArray*` that contained the `Port` information for both copies of the `TextDataBase` application.

However, if your application had filled in the `Port` information in the following way:

```
Port    myPort;

myPort.appName = ``TextDataBase'';
myPort.hostName = ``crow.stanford.edu'';
```

the Port Manager would return only information about the second `Port`, connecting your application to the second instantiation of the "`TextDataBase`" application since your application requested a "`TextDataBase`" from a specific host.

If the application you request is not currently registered with the Port Manager with with your application is communicating, `SenderGetPortFromName()` will attempt to launch the application for you. If successful, `SenderGetPortFromName()` will block for a few seconds while the new application launches, then will return the `Port` information for the newly-launched application. If unsuccessful, `SenderGetPortFromName()` will return a `NULL PortArray*` and an error code of `-1` as the return value of the function.

Note: `SenderGetPortFromName()` will not not try to launch an application on another machine. For example, if your application is running on host "`sioux`" sends the `SenderGetPortFromName()` message asking for an application running on "`crow`", `SenderGetPortFromName()` will not try to launch the new application on crow.

4 The Port Manager

The PortManager maintains a list of applications that have advertised themselves as alive and listening for messages from other applications. When your application creates a new Receiver object (by calling the NewReceiver() function), your application registers itself with the PortManager and thereby advertises its willingness to listen for messages.

The PortManager should always be running; ideally, it is started when the computer is turned on (usually the `/etc/rc.local` script will start the PortManager). There is exactly one PortManager per computer, and your application may register itself with any PortManager it deems necessary. For example, your application may be running on host "A", but it can register itself with the PortManager on host "B". How to do this will be explained later.

Applications should always create a Receiver. Think of the PortManager as directory assistance: creating a Receiver sends a message to the PortManager "publishing" your address should another application wish to communicate with you. When you make your address available to the PortManager, other applications can ask the PortManager for your address (as friends can call directory assistance to get your phone number). If your application does not register itself with the PortManager, other applications cannot find your address since the PortManager will have no record of it (just as directory assistance cannot give numbers not listed in the phone book).

Since the PortManager keeps a list of currently open applications, your application should be able to ask the PortManager for that list. The method `SenderGetOpenApps()` does exactly this: when your application calls

this method, the PortManager will give your application a list of all currently open applications that have registered themselves with that PortManager.

To find the address of a specific application, use the `SenderGetPortFromName()` method. You may be as specific or as general as you like when asking for an application's address; the amount of information you put in the Port passed into the function determines limits of the search.

For example, if your application wants to send messages to an application called "TextEditor" but doesn't care which host TextEditor is running on, the Port structure passed into `SenderGetPortFromName()` would be filled in as follows:

```
Port            myPort;
PortArray*      appsOpen;
int             result;
Sender*         applicationSender;

myPort.appName = "TextEditor";
myPort.hostName = AnyHost;
myPort.portNumber = 0;
result = SenderGetPortFromName(portMgrSender,&myPort,&appsOpen);
if (result != 0)        // non-zero means that the PortManager...
 exit(-1);              //...couldn't get an address for the...
                        //...information you gave it.

if (appsOpen->numberOfPorts > 0)
{
  myPort.hostName = appsOpen->portArray[0].hostName;
  applicationSender = NewSender(&myPort);
}
```

If your application wants to send messages to "TextEditor" but needs to speak specifically to the TextEditor running on the computer named "sioux", your application would use the same code as above, except that it would fill in the `myPort.hostName` field as follows:

```
myPort.hostName = "sioux";
```

Your application need not fill in the `portNumber` field before calling `SenderGetPortFromName()`, since applications almost always get their port numbers dynamically (that is, they listen on a different port number each time they are launched).

After you get the address of the application with which you wish to communicate, your application should now create a new Sender to send messages directly to that application, using the Port information returned by `SenderGetPortFromName()`. The last line of the sample code above illustrates this point.

5 Data Structures

There are seven data structures commonly used in an application in the **MAEstro** environment. They are:

1. Port
2. Selection
3. DispatchTable
4. Sender

5. Receiver
6. PortArray
7. IconData

Including the files <Sender.h> and <Receiver.h> will include the definitions for these data structures so that your application can use them.

The manipulation of the Sender and Receiver structures is all done through the methods described above; the programmer need know nothing more about the data internal to these two structures.

The use and manipulation of the Port structure is explained in section 3.8.

The use and manipulation of the DispatchTable structure is explained in section 3.5.

The use and interpretation of the Selection structure is explained in section 8.2.

6 Compilation

There are a few simple steps to follow when compiling applications for the **MAEstro** environment:

1. Define an environment variable NetworkSource to point to the directory in which the <Sender.h> and <Receiver.h> files reside.

   ```
   setenv NetworkSource /home/sioux/collab/Source/NetworkSource
   ```

2. If you are using a Makefile to compile your application, add the following definition to your CPPFLAGS variable, or whatever variable is appropriate on your system for adding include directories:

   ```
   CPPFLAGS += -I$(NetworkSource)
   ```

3. In the same Makefile, add the following to your LDFLAGS variable, or whatever variable is appropriate on your system for adding program libraries:

   ```
   LDFLAGS += $(NetworkSource)/libMAEstro.a
   ```

4. Be sure to link the Sun RPC library routines with your application; if you are using a Makefile, add the following to your LDLIBS variable or whatever variable is appropriate on your system:

   ```
   LDLIBS += -lrpcsvc
   ```

   If you are compiling your application manually (i.e., if you are not using a Makefile or other such tool), make sure that you link the **MAEstro** library **before** the Sun RPC library:

   ```
   $(NetworkSource)/libMAEstro.a -lrpcsvc
   ```

   Make sure that you are using an ANSI-compliant C compiler when you compile your application (the GNU C Compiler, gcc, is such a compiler). If your code is not ANSI-compliant, you may still use gcc to compile your code. To do so, put the following line in your Makefile:

   ```
   CC = gcc -traditional
   ```

   At this point, you should be ready to compile your application.

In our environment, we have chosen to link our applications with shared libraries instead of static linking. Because `gcc` does not support shared libraries, we do the link step with `cc`. To do this, define a rule in your Makefile as follows:

```
LINK.c    = cc $(CFLAGS) $(CPPFLAGS) $(LDFLAGS)
```

Notice that the link step explicitly uses `cc` instead of `gcc`. Using shared libraries will greatly decrease the size of your application binaries.

7 Programmer's Checklist

This section has yet to be completed.

8 Application Behavior

8.1 Key Concepts

The **MAEstro** environment assumes that applications in the environment have an internal notion of document, selection, and performance (including an estimated duration of the performance). Your application may interpret the notions of "selection" and "performance" any way it wishes, but the notion of a selection's performance duration has a concrete interpretation shared by all **MAEstro** applications, and is measured in milliseconds.

These three concepts are encapsulated in the following messages of the **MAEstro** protocol:

- **OpenDocument**
- **GetCurrentDocName**
- **GetSelection**
- **SetSelection**
- **PerformSelection**
- **HaltSelection**
- **PauseSelection**
- **ResumeSelection**

8.2 Document, Selection, and Performance

The best way to explain the interpretation of the concepts above is by pointing out how different applications would incorporate these concepts. The ability of applications to interpret **MAEstro** messages in whatever way suits them best is one of the advantages of the **MAEstro** protocol, giving the environment a great deal of flexibility.

- Consider a word processing application integrated into the **MAEstro** environment. The notion of a document for a word processor is obvious; a document is the paper, memo, letter, etc. typed in by the author. When the word processor receives an `OpenDocument` message from another application, the word processor would interpret the string passed to it as the name of a file to open. If such a file does not exist, the word processor would likely return an error code.

  For a word processor, the concept of a selection might mean the starting and ending bytes of a sequence of text within the current document. When the word processor receives a `SetSelection` message, the word

processor might try to highlight the range of bytes specified by the incoming message, within the current document. If there is no current document, the word processor would likely return an error code.

However, for a word processor, the concept of how long a selection might take to perform is less clear. A word processor doesn't normally associate time with the text in a document. However, since the protocol requires an application to return an estimate of how long the current selection will take to "perform", a word processor might return a hard-coded value of zero milliseconds (or whatever value seems suitable), or the word processor might use a nmemonic like "based on an average reading speed of 250 words per minute and the length of the current selection (25 words), the current selection should take an average of six seconds to read." The application can come up with the estimated time any way it sees fit (this is up to the application developer), but it should be a reasonable estimate if at all possible.

The notion of "performing" a selection is unclear for a word processor; the application developer might choose to show just the selected text in a separate window, or the application might simply show the selected text highlighted.

- Now consider an application that controls the operation a videodisc player that can search to any frame on the disc, play a single frame or any part of the video disc. The concept of a document may mean nothing to this type of application; in this case, the application could return `(char*)NULL` or a dummy document name like "Untitled". Whenever the application receives an `OpenDocument` message, the application would simply ignore it. In this case, your application probably doesn't need to override the default `OpenDocument` message handler.

  For this videodisc application, the notion of Selection might be defined as follows: the `start` and `end` fields of the `Selection` structure might contain the starting and ending frame numbers of a segment of video to play. The `duration` field would be the time necessary to play the segment. A still frame would be represented by the same value in both the `start` and `end` fields. In this case, the question of how long the frame lasts is up to the application developer; the application might decide that still frames should last for at least two seconds, or a still frame might only last as long as it would normally take to play (one thirtieth of a second, since there are thirty frames of video per second), or the application might allow the author to drag a knob specifying how long a still frame should last.

  The notion of performing a selection is clear for this application: performing a selection would mean playing the segment of video specified by the current selection. If the `PerformSelection` message is received but there is no current selection, the application might return an error code and do nothing else.

- There are many ways this core set of messages might be interpreted by an application. An application that uses edit lists as documents (e.g., for a video editing system that stores lists of start and stop points to be used later when recording onto another videotape) might use only the `start` field of the `Selection` structure as an index into the edit list. Another application might use a special document name (e.g., "CommandMode") that puts the application into "command mode"; any `SetSelection` messages that are received from this point on might be used to set hardware parameters for a device (e.g., setting the value of a `start` field of a `Selection` to 3 means to change the Volume setting of a CD player; the CD application would get the volume from the `end` field). Another application might ignore `GetSelection` and `SetSelection` message altogether and interpret `OpenDocument` and `GetCurrentDocName` messages as command sequences instead of filenames.

As long as the application can parse anything that it sends to the requester, any interpretation is possible. Assume that the calling application knows nothing about the data in the `Selection` structures and return values of `GetCurrentDocName`. In other words, an application will send a `GetCurrentDocName` message and a `GetSelection` message; later the same application will send a `OpenDocument` message with the string your application previously returned via `GetCurrentDocument`, then the application will send a `SetSelection` message using the same `Selection` data your application returned via the `GetSelection` message. The only data the remote application is safe to interpret is the `duration` field of `Selection` data.

8.3 Which Messages Are Sent When?

MAEstro was designed for the authorship of multimedia documents. In a typical authoring situation, an author will open your application to create documents for that medium (e.g., a word processor to create text documents, a video editor to create video edit lists, a music editor to create musical scores, etc.). After the application's current document has been saved, an authoring application will typically send pairs of messages. The first message will be the `GetCurrentDocName` message, asking your application for the name of the currently active and open document. The second message will be the `GetSelection` message, asking your application for the current selection within the current document. The author will then go to your application and select a different part of your application's currently opened document, then use the authoring application to send a new pair of `GetCurrentDocName` and `GetSelection` messages.

During the "playback" phase (when somebody is viewing a multimedia document), the authoring tool will typically send messages in groups of threes. The first message will be an `OpenDocument` message, asking your application to open a particular document named during the authorship phase. The second message will be `SetSelection`, asking your application to select part of the current document (ideally the document just opened as a result of the `OpenDocument` message). The third message will be a `PerformSelection` message, asking your application to do whatever it takes to "perform" the current selection within the currently active document.

Any message may be sent to your application from any other application at any time. Therefore, if your application will be providing its own message handling routines, you must make sure that they always return valid data that won't cause your application to crash when later given the same data. For example, if you application has just launched and another application sends a `GetCurrentDocName` message, your application might return "Untitled", a special name known by your application to mean "There is no current document; don't do anything when asked to open this document."

Messages will not necessarily come in the order described above, but the **MAEstro** protocol was designed with the above scenario in mind.

8.4 Synchronization Rules

In a typical multimedia document, an author might want to play audio and video from different sources together. This requires some sort of synchronization of applications, which may or may not be possible. However, applications in the **MAEstro** environment will help alleviate synchronization problems if they follow a few simple rules:

1. When your application receives a `SetSelection` message, it should do everything it needs to do to get that media segment ready to "perform". For example, when a videodisc player receives the `SetSelection` message it should spin up the disc and seek to the beginning frame of video. A word processor might jump to the page on which the selected text appears.

2. When a `PerformSelection` message is sent, your application should be ready to perform its selection immediately; this should already be the case if your application's `SetSelection` routine was written correctly.

3. Your application's `PerformSelection` routine should take as little time as possible, so it can return to the calling application immediately. For a videodisc application, this might mean to send the brief command "Play the segment I already loaded in your memory", which would return immediately.

When an authoring application is performing a segment of a multimedia document that contains segments from several applications at once, it should first send `OpenDocument` messages to each of the applications, waiting for each application to respond. Then the authoring application should send `SetSelection` messages to each application, waiting for each to sync their media and respond when ready. Then the authoring application will send `PerformSelection` messages; if the other applications follow the rules listed above, each application should respond quickly enough that all the media start at roughly the same time, close enough that the delay between start times will not be noticed by the author.

For some applications, it might be difficult to do much pre-processing work in the `SetSelection` routine. For example, a Mandelbrot generator might consider the generation of an image itself as the "performance". In such a

case, your application should be written so that the `PerformSelection` routine begins the performance and returns as quickly as possible, perhaps setting a flag indicating that the application's main event loop should be interrupted to do a little bit of computation. That way the `PerformSelection` message exits quickly and your application can still do work that might take a while to complete.

8.5 Handling Multiple Documents

Your application should be able to handle more than one open document at a time, with one of them being the currently active document. Your application should always keep track of the currently active document, if possible.

An authoring application may ask your application to open several documents, including documents your application already has opened. If your application receives a request to open a document that is already opened, your application need not re-open the document but should make that document the currently active document, so that proceeding `SetSelection`, `GetSelection`, and `PerformSelection` messages apply to that document.

If your application receives an `OpenDocument` message but the requested document is already opened and has been modified since it was opened, your application has several options:

1. Use the already opened and modified document for the proceeding Selection requests;
2. Clear the modified document and revert to the last saved version on disk;
3. Mark the already opened document with a different temporary name, then open the requested document from disk. This way, both the last saved version and modified versions are available to the author (one for playback and one for modification). This solution might be confusing to the author, however, so take extreme care if your application will use this technique.

The application should avoid prompting the author for action in this case, since it will likely upset playback of a multimedia document. If your application will be doing any error handling, it should do it silently and without intervention by the author.

8.6 Consider The Network Interface As Important As the User Interface

When designing your application, consider the network interface to be as important as the user interface. For example, there are at least two ways for a document to be opened: the author might type in the name of a document to open, and another application might request your application to open a document. The same concept applies to setting and performing a selection. This means that your routines for opening a document, selecting part of a document, and performing part of a document should be callable both from the network and interactively.

8.7 Handling Bad Messages From Other Applications

Your application should be able to deal with ill-formed messages from other applications. For example, if an application sends a `GetCurrentDocName` message and your application has no currently active document, your application should return an error code or a "dummy" name indicating that there is no currently opened document.

Your application should check `SetSelection` requests to see that the `start` and `end` fields are valid and that there is a currently active document for which the Selection applies.

When your application receives a `PerformSelection` message, it should check if there is something to perform. If there is nothing to perform, the application should return as quickly as possible.

In short, your application should be robust enough that it reacts gracefully to errors. It is critical in a real-time performance environment such as **MAEstro** that applications do not delay or stop a performance because of unrecoverable errors. Your application should do everything it can to insure that "the show must go on."

9 Sample Program

The following program is about the simplest program that can be written in the **MAEstro** environment. The purpose of the program is to show how an application will register itself with the PortManager.

This program first fills in a `Port` with information for the PortManager's address (host name and port number — the PortManager listens on a well-known port defined by the constant "PortMgrPortNumber"). The program then uses this information to create a new `Sender`, thereby creating a communication link with the PortManager.

If successful, the program will next try to register itself with the PortManager, using the `NewReceiver()` method. The `Sender*` passed in as argument is the `Sender*` used to communicate with the PortManager. The `NewReceiver()` call also creates a network connection on which the program would listen for incoming messages; however, the purpose of this sample program is to show how to create a connection with the PortManager, so this program does not contain code to listen for incoming messages.

If the call to `NewReceiver()` is successful, the program will immediately destroy the receiver using the `DestroyReceiver()` method. This method will inform the PortManager that the program is no longer advertising its services. The method also closes down the incoming message link, and frees space taken by the `Receiver*` passed in argument.

```
#include <stdio.h>
#include <Sender.h>
#include <Receiver.h>

/* Usage: connectWithPortMgrTest <hostName>
 *      where "hostName" is the name of the host on which the
 *      PortManager is running.
 */


main(int argc, char** argv)
{
  Sender*       sender;
  Receiver*     receiver;
  Port          senderPort;
  int           result;

  printf("The Port Manager should be listening on port number %d.\n",
         PortMgrPortNumber);
  senderPort.hostName = argv[1];
  senderPort.portNumber = PortMgrPortNumber;
  sender = NewSender(&senderPort);
  if (sender == (Sender*) NULL)
    exit(1);
  receiver = NewReceiver(sender,argv[0],AnyPort);
  if (receiver == (Receiver*) NULL)
    exit(1);
  printf("Connected with the Port Manager.\n");
  DestroyReceiver(senderPort, receiver);
  exit(0);
}
```

References

[Usenix] George D. Drapeau and Howard Greenfield, "MAEstro — A Distributed Multimedia Authoring Environment", *Proceedings of the 1991 Summer USENIX Conference*, Nashville, Tennessee, June, 1991.

[SunRPC] *Network Programming Guide*, Sun Microsystems, Inc., Mountain View, CA.

[NeXT] *NeXT System Reference Manual — Release 1.0 Edition*, NeXT Computer, Redwood City, CA.