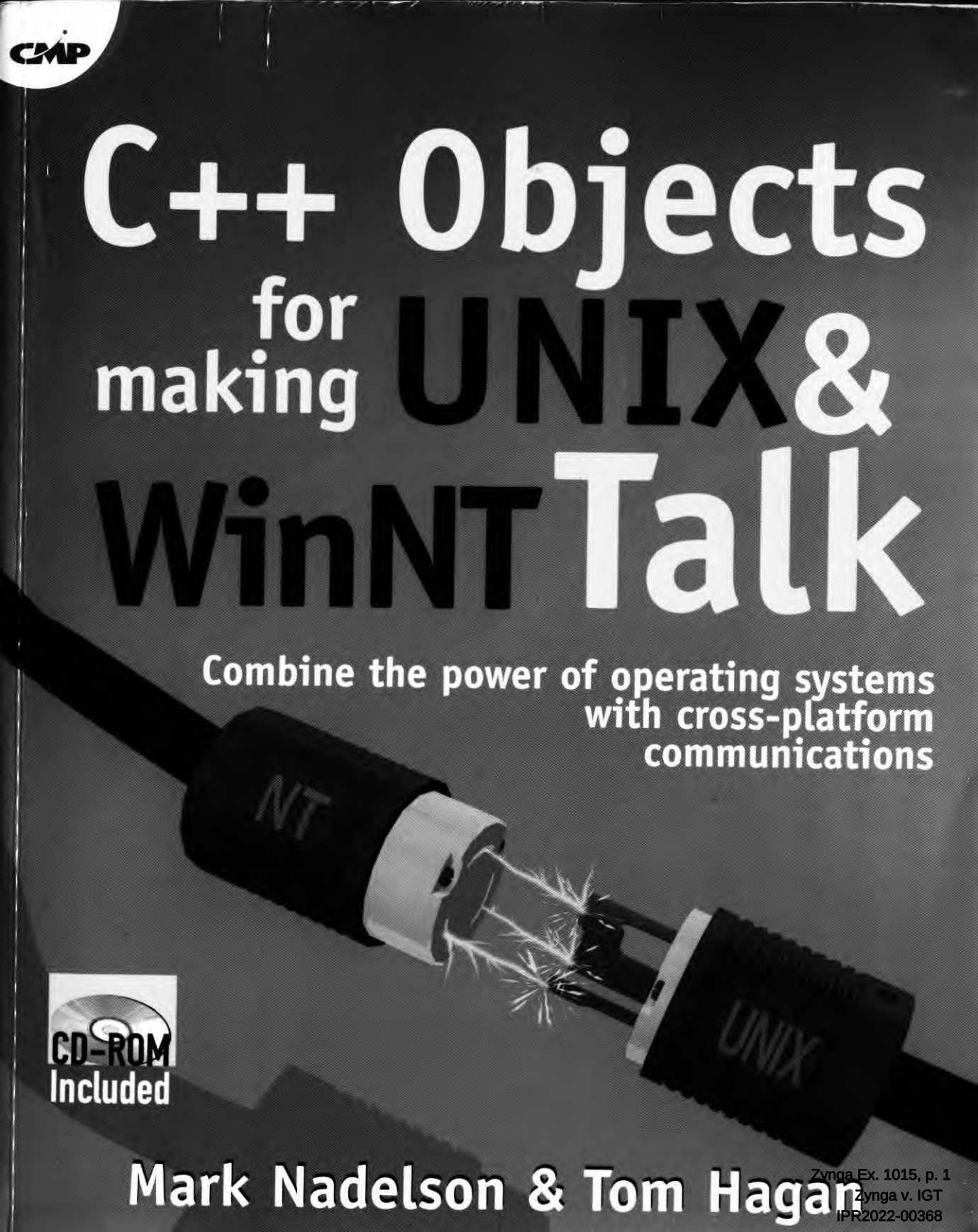
CMP

C++ Objects for making UNIX & WinNT Talk

Combine the power of operating systems with cross-platform communications



CD-ROM Included

Mark Nadelson & Tom Hagan

C++ Objects for Making UNIX and WinNT Talk

Mark Nadelson and Tom Hagan

CMP Books
Lawrence, Kansas 66046

CMP Books
CMP Media, Inc.
1601 W. 23rd Street, Suite 200
Lawrence, KS 66046
USA

Cover art created by Robert Ward.

Distributed in the U.S. and Canada by:
Publishers Group West
1700 Fourth Street
Berkeley, CA 94710
1-800-788-3123

ISBN: 1-929629-07-9

NETWORK PROGRAMMING

Want to reduce your overall cost of ownership? The object-oriented techniques for cross-platform communications presented in this book will enable you to develop applications capable of accessing functionality located on either UNIX or Windows NT.

This book delivers an in-depth look at cross-platform techniques beyond socket communications. It builds on the authors' earlier ti *Making UNIX and Windows NT Talk*, which demonstrated how to UDP and TCP socket communications to implement cross-platform tems in a variety of run-time environments.

***C++ Objects for Making UNIX and WinNT Talk* is a practical guid implementing object-oriented cross-platform communications.** Each chapter introduces a new concept, explained with supporting architectural drawings, C++ classes, and working applications. You learn how to create increasingly sophisticated C++ inter-platform objects that will enable you to implement cross-platform communications in a wide variety of architectures — single-threaded, multi-threaded, and Windows-based applications.

Learn how to use these advanced techniques that provide features not inherently available from pure sockets.

- **Remote Procedure Calls (RPC)** that call a procedure from an application running on one machine to execute on another.
- **Remote Execution (REXEC),** the daemon residing on UNIX workstations, provides a powerful communication channel requiring user authentication.
- **File Transfer Protocol (FTP)** with feature-rich custom file transfer systems.
- **Cross-Platform Semaphores** that can protect shared process resources, or facilitate multi-platform processes.
- **Shared Memory** processes that can write and read from a central resource independent of each other.
- **Pipes** for process-to-process communications that transmit and receive data using standard input and output file descriptors.
- **Publish and Subscribe** application frameworks that allow UNIX and NT to communicate without customizing message sending and translation.

C++ Objects that you can plug into your own applications are supplied on the companion CD-ROM.

CMP Books is an imprint of CMP Media, Inc.
Other fine publications in the CMP family include:

C/C++ Users Journal

Windows DEVELOPER'S JOURNAL

Sys Admin

CMP books
R&D Developer Series

experience in developing network software for telecommunications, digital signal processing, internet programming, and financial institutions. It is in their development of mission critical applications for financial institutions, where legacy UNIX systems are wedded to Windows NT workstations, that they have honed their skills in inter-platform communications.

$49.95 • ISBN: 1-929629-07-9

Table of Contents

8

Chapter 8

Cross-Platform Publish and Subscribe

The goal of this chapter is to provide an understanding of the generic messaging concept and architecture. In previous cross-platform communication systems, both the client and sever had knowledge of the data being communicated and its underlying structure. If the communication data structure changed, both the client and server had to be updated and recompiled to reflect that change. Generic messages solve the problem and complexity of synchronizing data structures among communicating processes. Section 8.1 introduces the concept of generic messages, and Section 8.3 explains how to develop a generic message framework.

Cross-platform communication using publish and subscribe The cross-platform communication method of publish-and-subscribe enables processes on both Windows NT and UNIX workstations to receive messages sent by other processes running on the two operating systems. Publish and subscribe has many advantages over other types of cross-platform communication, including the ability to send and receive generic messages. Section 8.2 introduces the publish-and-subscribe architecture and explains its advantages.

Creating a cross-platform publish-and-subscribe system In order to create a cross-platform publish-and-subscribe system, a Subscribe server must be developed. The Subscribe server is responsible for receiving subscriptions and publishing messages to the subscribed clients. The Subscribe server must be able to process published messages and requests for subscriptions from multiple executables at once. These messages can originate from processes running on two different operating systems. Section 8.4 explains how to create the Subscribe

server on both UNIX and Windows NT and Section 8.5 shows how to develop a cross-platform publish-and-subscribe system that passes generic messages.

8.1 Introduction

One of the greatest difficulties with cross platform communication is properly aligning data structures and adjusting for big endian/little endian differences. In previous chapters, these difficulties have been overcome by using one of two methods. The first method is to force all elements of the communicated data structure into character arrays. A character is a single byte and has no ordering issue. Both UNIX and Windows NT align character arrays to the nearest byte implicitly, correcting the byte alignment issue. Integrating multibyte data types such as integers within a data structure causes the resulting data structure to align differently depending on the operating system being used.

The second method used to overcome the structure alignment big endian/little endian problem is to encapsulate multibyte data types within the `XPCEndian` class (See Chapter 5 of *Making UNIX and Windows NT Talk*). The `XPCEndian` class converts multibyte data types into a character array. This can be done because a multibyte data type is stored in memory as consecutive bytes — exactly the way a character array is represented. Converting to character arrays avoids structure alignment issues. The `XPCEndian` class also contains methods for reversing the ordering of bytes so that endian issues are avoided.

Although the two methods — character arrays and the `XPCEndian` class — overcome the problems related to sending data structures between UNIX and Windows NT, a common complexity still exists. All methods of interplatform communication used thus far require customizing the client and server's method of sending data structures. Both applications need to know the data structures being sent and received and how to decode them once received. In previous applications, decoding transferred data structures was accomplished by encapsulating the transferred data within an object, whereby the object contained methods for correctly storing and extracting the data elements.

A maintenance issue arises when using the same class in two different places. If the class is changed on the client but not on the server, the system ceases to function correctly. The fix for this situation is to update the data class on the server side, immediately followed by an update on the client side. If only one side is updated, disastrous results may occur. The need to change the data transfer class on one system but not on the other is a realistic situation. For example, if the application that receives data needs to store information regarding the data's source, the data class storing the incoming message is a good place to put this. Unfortunately, increasing the size of the data class results in errors when attempting to transfer the same data object back to its source since the target of the transfer still maintains the original data structure.

Another method of interplatform communication overcomes these problems related to data transfer. The data structure alignment and big endian/little endian issues are avoided by incorporating character arrays and `XPCEndian` objects into the method. The issue regarding the maintenance of data transfer objects across platforms is also avoided through the use of a generic messaging framework. The generic messaging framework enables data objects to be transferred between client and server applications regardless of the type or amount of data. The creation and use of the generic messaging class framework are the focal points of this chapter.

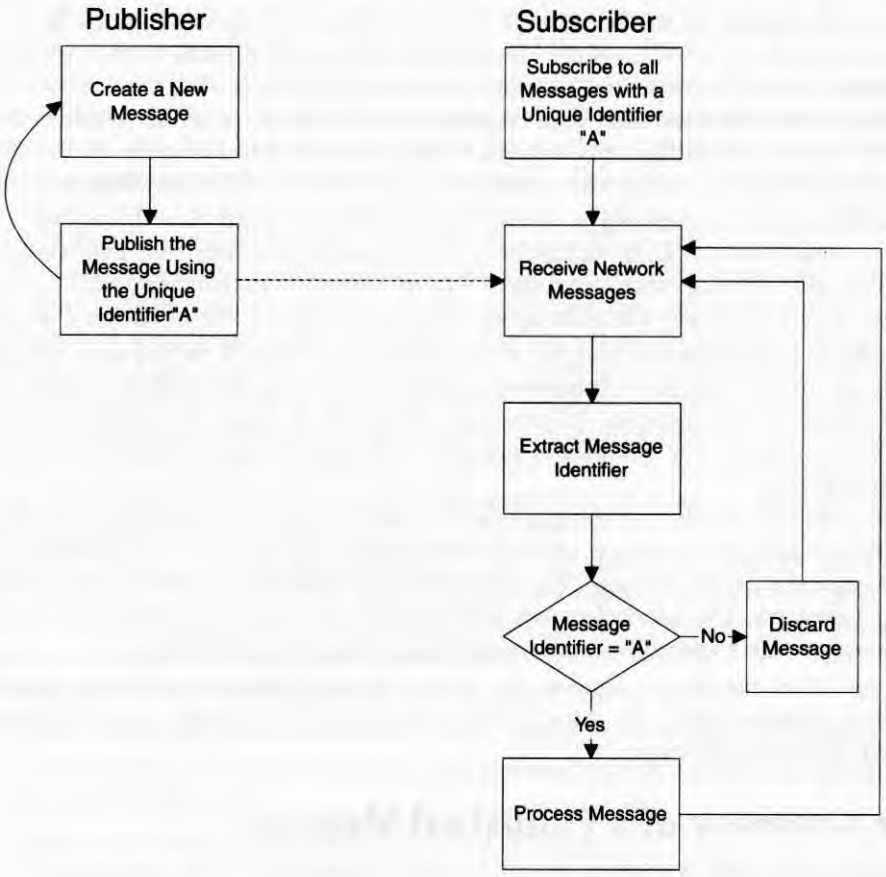
8.2 Publish and Subscribe

The generic messaging class framework uses the data transfer concept of publish and subscribe. The basis for a publish-and-subscribe system is as follows.

- Processes receiving network messages subscribe to all messages they want to receive. The subscription is accomplished by using a label that uniquely identifies the message being transferred.
- Processes sending network messages publish their messages using a unique identifier. If the process receives a message whose identifier matches the one subscribed to, the message is stored; otherwise, the message is discarded.

Figure 8.1 illustrates the basic architecture of a publish-and-subscribe system.

Figure 8.1 Architecture of a Publish-and-Subscribe System



The communicating processes in a publish-and-subscribe system are not directly connected to each other. This enables a subscriber to exist without a publisher and a publisher to exist without a subscriber. When a process subscribing to a message initializes, it waits for messages

bearing the identifier to which it has subscribed. If the process publishing such a message does not exist, the subscriber does not fail but continues to wait for the message to which it has subscribed. Likewise, if a publishing process has no subscribers, the publisher does not terminate but publishes the messages, which are sent over the network and discarded.

In previous systems, communication via TCP sockets fail if the connection with their communicating process is broken or does not exist at the time of data transfer. Transferring data via UDP sockets avoids the situation of having all communicating entities connected. Because the communicating processes are not connected, UDP communication avoids having all communicating processes initialized and running at the same time. Unfortunately, UDP sockets do not provide reliable data transfer. The publish-and-subscribe system eliminates the restriction of having all system components running and provides reliable data transfer when both the publish-and-subscribe processes are active at the same time. Providing reliable data transfer is discussed in Sections 8.4 and 8.5.

Removing the restriction that all communicating entities be active provides several advantages.

- It avoids system failure because of the loss of a communicating entity. In a system that communicates via TCP sockets, all clients are connected directly to their server. If the connection breaks because of a network failure or the loss of the client or server, the entire system fails when the next data transfer action takes place (either sending or receiving). This type of system failure is avoided within a publish-and-subscribe system since the processes publishing messages are not directly connected to the subscribing processes.
- Publishing and subscribing processes can be initialized at any time throughout the life cycle of a cross-platform communication system because processes that publish and subscribe do not rely on each other's existence. This means that within a cross-platform communication system, processes can start up and terminate without fear of harming the system.
- Multiple publishers and multiple subscribers can coexist. A publish-and-subscribe system enables many processes to publish a message using the same identifier and many processes to subscribe to a message. All processes subscribe to a specific message regardless of the message source. The complexity involved with connecting to all sources of a message is eliminated.
- An unlimited number of messages can be received from a single socket connection. Each message delivered within a publish-and-subscribe system contains a subject or identifier associated with the message. The subscriber can subscribe to an unlimited number of message subjects. The subscriber would have to know how to decode each message, but it would be able to receive all messages on a single socket connection.
- A publisher can also be a subscriber. A process publishing messages can also subscribe to other messages using the same socket connection. The process can even subscribe to its own published messages.

The Anatomy of a Published Message

A published message is made of up one or more components. Each component of the message is created and sent independently, and the entire message is reconstructed on the receiving end. The first component of all published messages is the subject component. The subject of a message identifies the type of message, and all messages bearing a particular subject name contain the same structure. The subject component also contains the total number of body

components. After decoding the component size of the message, the publisher and subscriber create message structures large enough to hold all components of the message and know how many components to receive. Each message component contains a unique identifier in which the identifier signifies the position of the message component within the entire message. This identifier can be used by the receiving process to reconstruct the message in its original format and to detect errors in message transmission. Transmission errors are detected by examining the identifiers received. If an identifier is missing or out of sequence, a message transmission error has occurred. The subject component's identifier is always 0. The structure for the subject component is shown in Figure 8.2.

The remainder of the message components makes up the body of the message. These components are referred to as body components. Each body component is independently created and added to the complete message. Body components are objects of one type but can contain data of any type and size. A single body component object is created so that all processes in a publish-and-subscribe system can receive and send messages regardless of the message composition. The objects that make up the body of the message contain a data member that is a large character array used to store data types in their string representation. Character arrays are sent over the network in their original form and do not need to be converted to a format specific to an operating system. The message component contains a flag to indicate the original type of the data stored within the character array. If the original type is a multibyte data structure, such as a `double` or an `int`, and the message component is received by a different operating system than that from which it originated, the value of the component is returned in its endian form. The operating system data member used in the subject component determines whether the component was received from a foreign operating system.

Each message component also includes a label, which uniquely names the component and is sent along with its value. The label is a character string and can be used to extract a particular element of the message. The advantage of using a label to retrieve message elements is that the receiving end does not need to know the ordering of body components; it must only know the names of the elements to extract. If the elements of a particular published message change, the subscribing process does not need to be altered or recompiled. It will only extract the components it requires, ignoring the rest. The ability to continue processing the same message bearing a specific subject holds true regardless of whether elements are added or deleted or their ordering is changed.

The final element of a body component is the component ID number. Each component contains a unique ID number. The component IDs are ordered in the sequence in which they are created and are attached to the complete message. The subject of a published message contains ID number 0. The component ID is used in several ways.

- The value of a body component can be extracted based on its ID.
- The subscribing process can determine if any message components are missing.
- The subscribing process can order the message components received in the same order as they were originally sent.

The structure of a message component is shown in Figure 8.3.

Figure 8.2 The Subject Component of a Published Message

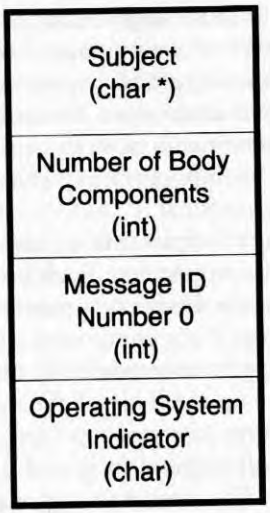
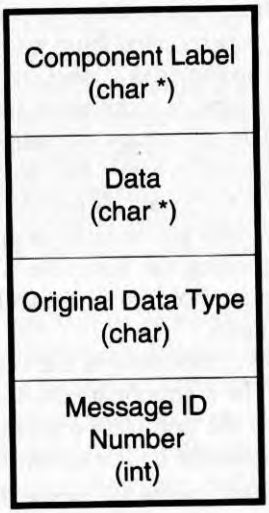


Figure 8.3 The Message Component Structure



How to Create, Publish, and Subscribe to Messages

When a message is created, a new message template object is instantiated. The message template object contains all components of the message, including its subject. The message template is created using a subject, its originating operating system, and the number of body components. Once the template object is created, each body component is inserted. A body object is inserted using an object label and value. The label uniquely defines the body component, and the value can be made of any base data type, including

- `int`,
- `double`, or
- `char *` (string).

When the message template object is completed, it contains a subject component and the number of body components specified. The message is sent to the subscriber(s) one component at a time. The first component sent is the subject. The subject informs the subscriber of the type of message being sent and the number of body components that complete the message. If the receiving process has not subscribed to the published message, it discards it and waits for the next message. If it has subscribed to the message, it decodes the subject component and creates a message template object to hold the number of body components specified in the subject. The receiving process loops until it receives all body components and inserts each component into its message template object. The body components are added to the message template in the order dictated by the component's ID. If an ID is missing, the message is corrupt and it is discarded. When all components are received, the message can be decoded.

The receiving process has two choices of how to decode the message: by label or by ID number. When all the needed components of the message have been extracted, the message is discarded. Figure 8.4 illustrates the steps required to create, receive, and decode a generic message.

8.3 Creating the Generic Message Class

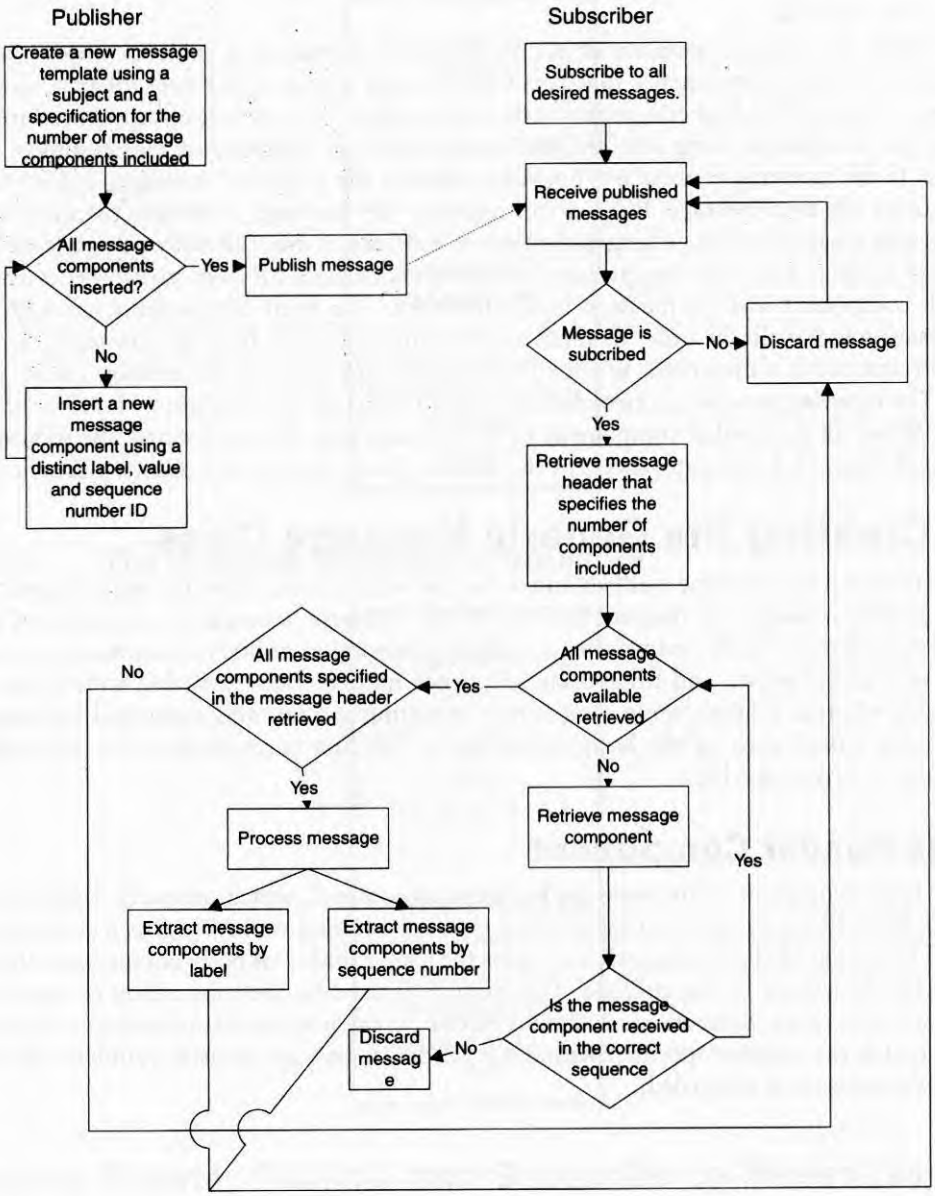
The first step in creating a publish-and-subscribe system is to define the objects used to create the generic message. As describe in Section 8.2, a generic message is composed of a header and zero or more body components. A subject, the number of body components, the originating operating system, and the message sequence number `0` are included within the message header, whereas a label, value, originating operating system, and sequential message ID are included within each of the body components. The first component of the message — the header — is described first.

The Header Component

The first component of the message header is the subject, which uniquely defines a message type. Subscribing processes subscribe to a message's subject and store it as a character string.

The header of the message also includes the total number of body components that constitute the remainder of the message. This number is used by the subscribing process so that it knows how many components it should receive. If the number of components received does not match the number specified within the header, a message transfer problem has occurred and the message is discarded.

Figure 8.4 Creating, Receiving, and Decoding a Generic Message



The number of messages is stored as an integer within the header component. The representation for the number of body components must be stored in its original and endian formats, because the message may be transferred to and from heterogeneous or homogeneous operating systems. The integer is stored as an `XPCEndian` object, which automatically stores the integer in two formats. A problem arises in choosing the correct integer representation when the subscriber extracts the number of body components from the header. This is resolved by having the publisher specify its operating system within the header and having the subscriber specify its operating system when extracting the number of body components. If the two operating system specifications differ, the endian value is returned; otherwise, the original value is returned. The operating system value is determined at compile time using the compiler definition `UNIXOP` when compiling on UNIX and `WINDOWS_NTOP` when compiling on Windows NT. Even though the operating system is defined at compile time, it can still be overridden when calling functions that return a multibyte data type.

Each component that composes a published message contains a sequential ID number. The ID number for the header component is always `0`. If the first message component a subscriber receives does not have a sequence number of `0`, the remainder of the message is discarded.

The header component has the additional task of defining the type of request being made: `PUBLISH` or `SUBSCRIBE`. If the request type is `SUBSCRIBE`, the number of body components is `0`, since the message only includes a header. The header component is defined in the `XPCHeader` class (Listing 8.1).

Listing 8.1 The `XPCHeader` Class Definition

```
#include <string.h>
#include <XPCEndian.h>   // Defines the XPCEndian class
#include <stdlib.h>

// Message header type definitions
#define PUBLISH 0
#define SUBSCRIBE 1

// The operating system definition is determined at compile time.
#ifdef UNIX
   #define UNIXOP 0
   #define OPSYSTEM UNIXOP
#else
   #define WINDOWS_NTOP 1
   #define OPSYSTEM WINDOWS_NTOP
#endif

class XPCHeader
{
   char cPubSubType;     // Stores the message header type
```

```
    char cOpSystem;       // Stores the operating system definition
    char sSubject[256];   // Stores the subject of the message
    XPCEndian<int> iNumComponents;   // Stores the number of body components
                                     // that make up a message
public:
    // Default constructor. Initializes the number of body components to 0. This
    // constructor is used when the header is received from the network. The
    // contents of the received message header replaces the default constructed
    // message header.
    XPCHeader() { iNumComponents = (int)0; }

    // Constructor. Defines the originating operating system, the publish type,
    // the message subject, and the number of body components
    XPCHeader(char _cType, char *_sSubject, int _iNumComponents = 0,
              char _cOpSystem = OPSYSTEM)
    {
       cOpSystem = _cOpSystem;
       cPubSubType = _cType;
       strcpy(sSubject, _sSubject);
       iNumComponents = _iNumComponents;
    }

    // Method that is used to set the private data members.
    void vSetValues(char _cPubSubType,
                    char *_sSubscription,
                    int _iNumComponents = 0,
                    char _cOpSystem = OPSYSTEM)
                    {
                      cOpSystem = _cOpSystem;
                      cPubSubType = _cPubSubType;
                      strcpy(sSubscription, _sSubscription);
                      iNumComponents = _iNumComponents;
                    }

    // The number of body components is returned. The format of the return
    // depends upon the original operating system specification and the
    // operating system used to retrieve the count
    int iGetNumComponents(char _cOpSystem = OPSYSTEM)
    {
```

```
        // If the operating systems differ, the "endian" format is returned.
        // otherwise the original format is returned.
        if (_cOpSystem != cOpSystem)
            return iNumComponenets.getEndianValue();
        else
            return iNumComponents.getOriginalValue();
    }

    // The subject of the message is returned
    char *sGetSubject() { return sSubject; }

    // The message header function (publish or subscribe) is returned.
    char cGetPubSubType() { return cPubSubType; }
};
```

The Body Component

Messages that include data contain one or more body components. A body component comprises a value, a label that identifies the component, and a sequence number ID. The component's value can be of any basic type (a basic type is just a series of contiguous bytes). The component's value is stored as a byte array, since a byte array can be used to store any basic type. When a value is stored within a body component, its endian state is also stored. The endian value is stored within another private data member of the body component class and is the same as the original value, except that its bytes are reversed. If the value being stored is a string, its endian value is not stored.

Following the header component are zero or more body components. Body components are not necessary when creating and sending a generic message in two situations. The first is when the message is used for subscription purposes. In this case, the only component needed is the header component. The header component contains the name of the message being subscribed to and a message header of type SUBSCRIBE. The number of body components specified within the header is 0. The second situation in which body components may not be needed is when the message being published is used as an *event signaler*, which is a catalyst message that contains no information but, when received, causes specific actions within the receiving process to occur. Generic messages provide a flexible means for creating an event signaler because the receiving process can perform different tasks based on the subject of the published message.

The original type of the value is also stored within the body component object. The original type is stored as a single byte and defined using unique integer values. Storing the original type enables the body component class to know how to return the value when it is requested. The type definition can be extracted and used by the subscribing process. When used this way, the value of the component can be retrieved and placed in a variable based on the component's originating type.

Another value required by the body component class is the operating system. This value is stored as a private data member and transferred through message publication. The operating

system is determined at compile time but can be overridden by calling methods that return data composed of multibyte types. The originating operating system is used when retrieving the value from the body component. If the retrieving operating system differs from the originating operating system, the endian representation of the original value is returned; otherwise, the original value is returned.

Values stored within a body component are associated with two identifiers. One identifier is a unique string that is used as the body component's label. Associating a label with a component makes the retrieval of the body component independent of its position. If ordering of body components changes because of the addition or deletion of other components, the subscriber can still access the specific value using the associated label.

The other body component identifier is a sequence number ID. This ID serves two purposes. Each time a body component is added to the message, it is given an ID that is in sequence with the previous component. The header component is always the first component sent and contains an ID of 0. The first body component has an ID of 1 and the remainder of the body components increment accordingly. The ID is used by the subscriber to ensure that all components of a message are received. If an ID is missing, the message is discarded. If for some reason the components are received out of sequence, the subscribing process discards the message and assumes a network transport error has occurred. The ID can also be used to retrieve body component values. Requesting a specific sequence number from the array of body components can retrieve a value. The ID is stored as a `XPCEndian` object so that its value can be extracted correctly when sent from a foreign operating system. When the ID is retrieved, the operating system must be specified so that the correct format of the ID can be returned. The `XPCComponent` class is shown in Listing 8.2.

Listing 8.2 The `XPCComponent` Class Definition

```
#include <XPCEndian.h>      // Defines the XPCEndian class
#include <XPCHeader.h>      // Defines the XPCHeader class
#include <XPCTcpSocket.h>   // Defines the XPCTcpSocket class

// Original type definitions
#define STRING 0
#define INT 1
#define DOUBLE 2

class XPCComponent
{
   char sValue[512];       // Byte representation of the component's value
                           // in its original format
   char sEndianValue[512];   // Byte representation of component's value
                             // in its endian format
   char sLabel[512];       // The label that uniquely identifies the component
   XPCEndian<int> iId;     // The sequence ID the uniquely identifies the
                           // component
```

```
    char cOpSystem;       // The operating system used when storing the
                          // component's value
    char cType;           // The original type of the component

    // Private member function used to convert the value stored in sValue to its
    // endian representation (sEndianValue).  This member function is called in
    // methods used to store the original value.  See the CD-ROM for the
    // vSetEndianValue() listing.
    void vSetEndianValue(int _iSize);
public:
    // Default constructor.  Sets all data members to NULL
    XPCComponent()
    {
       memset(this, 0, sizeof(XPCContainer));
    }
    // Constructor. Sets all data members to NULL and sets the operating
    // system that is used to store values
    XPCComponent(char _cOpSystem = OPSYSTEM)
    {
       memset(this, 0, sizeof(XPCContainer));
       cOpSystem = _cOpSystem;
    }

    // The vSetValue() member function is overloaded to store all basic data
    // types. The value, its label and Id are stored.

    // Stores values that are integers
    void vSetValue(char *_sLabel, int _iValue, int _iId)
    {
       // The original value is stored within the sValue byte array
       memcpy((void *)sValue, (void *)&_iValue, sizeof(int));

       // The endian value is stored within the sEndianValue byte array
       vSetEndianValue(sizeof(int));

       strcpy(sLabel, _sLabel);   // The value's label is stored
       cType = INT;               // The original type of the value is stored
       iId = _iId;                // The ID of the component is stored
    }
```

```
// Stores values that are doubles
void vSetValue(char *_sLabel, double _dValue, int _iId)
{
   // The original value is stored within the sValue byte array
   memcpy((void *)sValue, (void *)&_dValue, sizeof(double));

   // The endian value is stored within the sEndianValue byte array
   vSetEndianValue(sizeof(double));

   strcpy(sLabel, _sLabel);   // The value's label is stored
   cType = DOUBLE;            // The original type of the value is stored
   iId = _iId;                // The ID of the component is stored
}

// Stores values that are strings
void vSetValue(char *_sLabel, char *_sValue, int _iId)
{
   // The original value and endian value are stored in their original
   // formats
   strcpy(sValue, _sValue);
   strcpy(sEndianValue, _sValue);

   strcpy(sLabel, _sLabel);   // The value's label is stored
   cType = STRING;            // The original type of the value is stored
   iId = _iId;                // The ID of the component is stored
}
// Retrieves the components original type
char cGetType() { return cType; }

// The iGetValue() methods retrieve the component's value and stores it in
// the variable that is passed in. The format of the value stored is based
// on the operating system specified in the cOpSystem data member and the
// operating system passed as a parameter. If data type used to store the
// component's value is not the same as the original type (cType),
// iGetValue() returns FALSE(0), otherwise TRUE(1) is returned.

// Returns the component's value as an integer.
int iGetValue(int *_iRetval, char _cOpSystem = OPSYSTEM)
```

```
{
   if (cType != INT)
      return 0;

   if (cOpSystem != _cOpSystem)
      memcpy((void *)_iRetval, (void *)sEndianValue, sizeof(int));
   else
      memcpy((void *)_iRetval, (void *)sValue, sizeof(int));
   return 1;
}

// Returns the component's value as a double.
int iGetValue(double *_dRetval, char _cOpSystem = OPSYSTEM)
{
   if (cType != DOUBLE)
      return 0;

   if (cOpSystem != _cOpSystem)
      memcpy((void *)_dRetval, (void *)sEndianValue, sizeof(double));
   else
      memcpy((void *)_dRetval, (void *)sValue, sizeof(double));
   return 1;
}

// Returns the component's value as a string
int iGetValue(char *_sRetval, char _cOpSystem = OPSYSTEM)
{
   if (cType != STRING)
      return 0;

   strcpy(_sRetval, sValue);
   return 1;
}

// Returns the component's label
char *sGetLabel() { return sLabel; }

// Returns the component's ID
int iGetId(char _cOpSystem = OPSYSTEM)
```

```
    {
        if (cOpSystem != _cOpSystem)
            return iId.getSwapValue();
        else
            return iId.getValue();
    }
};
```

Putting the Message Components Together in XPCMessage

The message components are stored together within an XPCMessage object, which creates the header component and body components. XPCMessage contains private data members that include XPCHeader and a pointer to the array XPCComponent. Also included as private data members is the total number of XPCComponent objects (iNumComponents) and the current body component ID. The current body component ID is stored in the iCurrentComponent data member, is initialized to 0 at the time of XPCMessage construction, and is incremented each time a new XPCComponent is inserted. XPCMessage contains methods that allow the insertion and retrieval of body components. When inserting an XPCComponent, the iCurrentComponent data member is passed to XPCComponent's constructor and stored as the object's ID. Both the iCurrentComponent and iNumComponents data members are integers and stored as XPCEndian objects so that they can be extracted when sent to a different operating system.

In order for the receiving process to know which format to use to extract the values of iNumComponents and iCurrentComponent, it must know the operating system used to store their values. This is accomplished by storing the operating system value in the private data member cOpSystem. The operating system values are defined by #define macros and are set at compile time using the precompiler definitions of UNIXOP or WINDOWS_NTOP. The operating system chosen at compile time is the default when storing data in cOpSystem, but it can be overridden during XPCMessage construction.

XPCMessage is responsible for sending and receiving its header and body components. It sends and receives components using an XPCTcpSocket object passed during construction. A pointer to the XPCTcpSocket object is stored as a private data member within XPCMessage.

The final data member of XPCMessage dictates its behavior. XPCMessage serves two purposes: It can be used to construct and publish a message, or it can be used to subscribe to a published message. The publish-and-subscribe indicators are defined using #define macros and are passed in at the time of XPCMessage construction. The indicator is stored within the cType private data member.

Besides creating messages and publishing and retrieving messages, other methods within XPCMessage include extracting the values of the private data members, extracting the values of body components either by label or by ID, and inserting body components. Detailed descriptions of XPCMessage (Listing 8.3) member functions are described throughout the remainder of this section.

Listing 8.3 The XPCMessage Definition

```
#include <XPCEndian.h>          // Defines the XPCEndian object
#include <XPCTcpSocket.h>       // Defines the XPCTcpSocket object
#include <XPCHeader.h>          // Defines the XPCHeader object

class XPCMessage
{
   XPCEndian<int> iNumComponents;      // The total number of body components
                                       // included.
   XPCEndian<int> iCurrentComponent;   // The current body component ID.
   char cType;              // The XPCMessage object type (PUBLISH or
                            // SUBSCRIBE).
   char cOpSystem;          // The operating system indicator.
   XPCHeader Header;        // The header component object.
   XPCComponent **ComponentList;   // The list of body components.
   XPCTcpSocket *Socket;           // The TCP socket used to receive and send
                                   // XPCMessage objects.
public:
   // Constructor used when creating a XPCMessage object for publishing a
   // message and sending out notice of message subscription
   XPCMessage(char *_sSubject, char _cType, XPCTcpSocket *_Socket,
              int _iNumParts = 0, char _cOpSystem = OPSYSTEM);

   // Constructor used for retrieving and storing a XPCMessage object sent
   // over the network
   XPCMessage(int _iNumParts, XPCTcpSocket *_Socket,
              char _cOpSystem = OPSYSTEM);

   // Destructor used to delete the ComponentList array
   ~XPCMessage() { delete [] ComponentList; }

   // Overloaded methods used to insert XPCComponent objects
   void vInsert(char *_sLabel, int _iValue);
   void vInsert(char *_sLabel, double _dValue);
   void vInsert(char *_sLabel, char *_sValue);

   // Overloaded methods to retrieve XPCComponent values based on their label
   int iGetValueByName(char *_sLabel, char *_sValue, char _cOpSystem);
   int iGetValueByName(char *_sLabel, int *_iValue, char _cOpSystem);
```

```
    int iGetValueByName(char *_sLabel, double *_sValue, char _cOpSystem);

    // Overloaded methods to retrieve XPCComponent values based on their ID
    int iGetValueByNumber(int _iID, char *_sValue, char _cOpSystem);
    int iGetValueByNumber(int _iID, int *_iValue, char _cOpSystem);
    int iGetValueByNumber(int _iID, double *_dValue, char _cOpSystem);

    // Retrieves the label associated with a body component given its ID
    char *sGetLabelByNumber(int _iID, char _cOpSystem);

    // Retrieves data type associated with value stored within an XPCComponent
    // object. Data type can be retrieved using body component's label or ID

    int iGetTypeByNumber(char &_cType, int _iLocation, char _cOpSystem);
    int iGetTypeByName(char &_cType, char *_sLabel, char _cOpSystem);

    // Retrieves the number of body components contained within the XPCMessage
    // object
    int iEntries(char _cOpSystem);

    // Retrieves the XPCComponent object based on its ID or label
    XPCComponent *GetComponent(int _iID);
    XPCComponent *GetComponent(char *_sLabel);

    // Methods used for publishing, subscribing and receiving XPCMessage objects
    void vPublish();
    void vSubscribe();
    static void vGetMessages(XPCTcpSocket *_Socket, XPCHeader &_Header,
                             void vCallBack(XPCTcpSocket *, int, char *),
                             char _cOpSystem);
};
```

XPCMessage Constructors

When the XPCMessage object is constructed, the header information is passed in as parameters to the vSetValues() method of XPCHeader. The header information includes the number of body components required when constructing XPCMessage, as well as XPCHeader. XPCHeader stores this number, and XPCMessage uses it to allocate memory for the array of XPCComponent objects. The XPCComponent array is initialized to the size specified.

XPCMessage also contains a private data member that stores how the XPCMessage object is used. This method can be either PUBLISH or SUBSCRIBE and is passed in as a parameter. If the

method chosen is SUBSCRIBE, the number of body components is 0, since no data is associated with a subscription other than the message that is subscribed. When a subscription message is defined, the number of body components need not be specified because its parameter value defaults to 0. The indication of PUBLISH or SUBSRIBE is stored within the XPCMessage object and passed to the XPCHeader object.

The body component ID, iCurrentComponent, is initialized to 0, and the operating system used to store XPCMessage values is also stored at this time. The operating system is defaulted to that chosen at compile time but can be overridden by specifying its value during construction.

Also required when constructing XPCMessage (Listing 8.4) is the TCP socket used to retrieve and send the object. The TCP socket is passed in as a pointer to an XPCTcpSocket object stored within a private data member, and it is used by other XPCMessage methods for sending and receiving.

Listing 8.4 The XPCMessage Constructor

```
XPCMessage::XPCMessage(char *_sSubject, char _cType, XPCTcpSocket *_Socket,
                       int _iNumParts, char _cOpSystem)
{
   cOpSystem = _cOpSystem;   // The operating system used is stored
   cType = _cType;           // The XPCMessage type (PUBLISH or SUBSCRIBE) is
                             // stored

   iCurrentComponents = 0;         // The current component ID is stored
   iNumComponents = _iNumParts;    // The total number of components is stored
   Socket = _Socket;         // The XPCTcpSocket object used to receive and
                             // send the XPCMessage object is stored

   // The header component is initialized with values that include the type of
   // message being sent (PUBLISH or SUBSCRIBE), the subject of the message,
   // the number of body components, and the operating system used.
   Header.vSetValues(_cType, _sSubject, _iNumParts, _cOpSystem);

   // Array of body components is initialized to size specified by _iNumParts
   ComponentList = new XPCComponent*[_iNumParts];
}
```

The second purpose for which the XPCMessage object can be constructed is to capture incoming messages. When the message is transferred from one process to another, the components of the message are sent separately and reassembled into an XPCMessage object. The second XPCMessage constructor builds its full message from components received over the network. When the XPCHeader object is received, the number of body components specified is extracted. This number is passed to the XPCMessage constructor along with a pointer to the XPCTcpSocket used to retrieve the message and the message name. The constructor's final

parameter is the operating system that retrieves the data, which is set at compile time but can be overridden. The message name and number of components are used to set values within the XPCMessage XPCHeader object. The number of body components retrieved from XPCHeader is used to initialize the array of XPCComponent objects, and each body component is received using the XPCTcpSocket pointer.

When using this constructor within a Windows application that incorporates asynchronous sockets, iRecieveMessage() throws an exception because it is a blocking operation used within a nonblocking application. To overcome this problem, iRecieveMessage() is placed within a loop and each time an exception due to blocking is thrown, the process sleeps 100 microseconds and tries again. Sleeping helps the problem, since the exception is thrown because there is no data currently available on the socket. Sleeping gives the network time to send and receive the data.

The body components are placed into the XPCMessage array which is initialized with XPC-Component objects, and each is checked to make sure its ID is in the proper sequence. Two methods determine whether a message transfer error has occurred. The first is message sequencing. The order in which the body components are sent is the order in which they should be retrieved because of the method of network communication used. If a component is received out of sequence, a transmission error has occurred and the message is discarded. The second method of message transfer error checking is to check the total number of message components received. If, after all the components are received, the number received does not equal the value contained within XPCHeader, a transmission error has occurred and the message is discarded. Only if the correct number of XPCComponent objects are received, and received in the correct order, does the XPCMessage constructor return successfully. The constructor used to receive and store XPCComponent objects sent over the network is shown in Listing 8.5.

Listing 8.5 The Second XPCMessage Constructor

```
XPCMessage(int _iNumParts, XPCTcpSocket *_Socket, char *_sSubscription,
           char _cOpSystem)
{
   // The operating system value is stored so that proper "endian" translation
   // of multi-byte data types can occur
   cOpSystem = _cOpSystem;

   // Values are stored within the XPCHeader object
   Header.vSetValues(PUBLISH, _sSubscription, _iNumParts);

   // The number of body components to be received is initialized
   iNumComponents = _iNumParts;

   // The XPCTcpSocket object used to retrieve the body components is stored
   Socket = _Socket;
```

```
    // The array of XPCComponent objects is initialized to the size specified
    ComponentList = new XPCComponent*[iNumComponents];

    // The number of body components specified is retrieved
    for (int iCount = 0; iCount < iNumComponents; iCount++)
    {
       // The current element of the XPCComponent array is initialized
       ComponentList[iCount] = new XPCComponent();

       // The body component is received from the network and stored within the
       // current XPCComponent element.  The MSG_WAITALL option is specified to
       // make sure that the entire XPCComponent object is received
#ifdef UNIX
            Socket->iRecieveMessage((void *)ComponentList[iCount],
                                    sizeof(XPCComponent),
                                    MSG_WAITALL);
#else
      // If this is a Windows NT application loop until the data is available
      // on the socket
      while(1)
      {
         try
         {
            Socket->iRecieveMessage((void *)ComponentList[iCount],
                                    sizeof(XPCComponent), MSG_WAITALL);
         }
         catch(XPCException &exceptOb)
         {
            // If the exception is thrown due to a blocking process
            // within a non-blocking application, a sleep is done to
            // wait for socket data to arrive
            if (WSAGetLastError() == WSAENONBLOCKING)
            {
               Sleep(100);
               continue;
            }
            else
               throw exceptOb;
```

```
            }
            break;
        }
#endif
        // If the current element received is not equal to the ID stored within
        // the body component a transmission error occurred, and an exception
        // is thrown
        if (iCount != ComponentList[iCount]->iGetId(_cOpSystem))
        {
            XPCException newExcept("Received message out of sequence");
            throw newExcept;
        }
    }

    // If total number of body components received does not equal total number
    // specified, a transmission error occurred and an exception is thrown
    if (iCount != iNumComponents.getValue())
    {
        XPCException
            newExcept("Received the incorrect number of message components");
        throw newExcept;
    }
}
```

Inserting XPCComponent Objects

When body components are added, the overloaded `vInsert()` methods associate a label with a value and store it within the current element of the `XPCComponent` array. The label associated with the `XPCComponent` object should be unique to `XPCMessage`, since the first matching `XPCComponent` object is used when extracting values by label. `XPCMessage` stores the sequential ID that is associated with the inserted `XPCComponent` value. Each time a body component is inserted, the ID is passed to the `XPCComponent` object then incremented by one. The ID serves two purposes. It can be used to extract a particular element from the `XPCComponent` array and to check the order which the `XPCComponent` objects were received. Only one `vInsert()` method (Listing 8.6) is shown. The others contain the same code with different parameters.

Listing 8.6 The XPCMessage Overloaded vInsert() Method

```
void vInsert(char *_sLabel, int _iValue)
{
    // Current CComponent element is initialized. Note that the current ID is
    // retrieved using the getValue() method of XPCEndian. This method is used
```

```
    // since the operating system used to create the XPCMessage object is that
    // used to store the XPCComponent elements
    ComponentList[iCurrentComponents.getValue()] = new XPCComponent();

    // The body component's label, value , and ID are stored
    ComponentList[iCurrentComponents.getValue()]->vSetValue(_sLabel,
                                                _iValue,
                                                iCurrentComponents.getValue());

    // The current ID is retrieved, incremented by 1, and stored
    int iCount = iCurrentComponents.getValue() + 1;
    iCurrentComponents = iCount;
}
```

Retrieving XPCComponent Values

When an XPCMessage object is received, its body components are stored within the ComponentList private data member array. Two overloaded methods are provided to extract the values from a specified element of ComponentList. One method extracts XPCComponent values by ID. This method is know as iGetValueByNumber(), and it stores the specified XPCComponent value within the parameter provided. The extraction and storage of the XPCComponent value is determined by the data type of the parameter. If the parameter is an integer, the value stored within the XPCComponent object is stored as an integer. The XPCComponent value is stored if the original XPCComponent type specified matches the parameter's type. If the types do not match, iGetValueByNumber() returns 0, indicating failure; otherwise, iGetValueByNumber() returns 1, indicating success. iGetValueByNumber() also returns 0 if the ID specified does not exist. The iGetValueByNumber() methods are overloaded in order to store data of different types. Only one method is shown (Listing 8.7).

Listing 8.7 The iGetValueByNumber() Method

```
int iGetValueByNumber(int _iID, double *_dValue, char _cOpSystem)
{
   int iNum;

   // If the operating system used to retrieve the XPCComponent value differs
   // from the operating system used to store its value, the "endian" value of
   // the data member is returned
   if (cOpSystem != _cOpSystem)
      iNum = iNumComponents.getEndianValue();
   else
      iNum = iNumComponents.getOriginalValue();
```

```
    // If the ID specified does not exist, 0 is returned
    if ((_iID < 0) || (_iID >= iNum))
        return 0;

    // Components value is returned in the parameter passed in. If the original
    // type of the component was not a double, 0 is returned indicating failure;
    // otherwise, 1 is returned indicating success
    return ComponentList[_iID]->iGetValue(_dValue, _cOpSystem);
}
```

If the original type of the `XPCComponent` object specified is unknown, it can be extracted using the `iGetTypeByNumber()` method (Listing 8.8). The `XPCComponent` type is returned based on the element specified. If the element specified does not exist, `iGetTypeByNumber()` returns 0, indicating failure; otherwise, 1 is returned and the type is stored within the `_cType` parameter.

Listing 8.8 The `iGetTypeByNumber()` Method

```
int iGetTypeByNumber(int _iID, char &_cType, char _cOpSystem)
{
    int iNum;

    // If the operating system used to retrieve the XPCComponent value differs
    // from the operating system used to store its value, the "endian" value
    // of the data member is returned
    if (cOpSystem != _cOpSystem)
        iNum = iNumComponents.getEndianValue();
    else
        iNum = iNumComponents.getOriginalValue();

    // If the ID specified does not exist, 0 is returned
    if ((_iID < 0) || (_iID >= iNum))
        return 0;

    // The type stored within the XPCComponent element is stored within
    // the _cType parameter
    _cType = ComponentList[_iLocation]->cGetType();
    return 1;
}
```

Using `iGetTypeByNumber()`, a generic routine can be written that extracts all components of an `XPCMessage` object, regardless of type (Listing 8.9).

Listing 8.9 Extracting `XPCMessage` Components Regardless of Type

```
char cType;          // Stores the component's data type
char sString[256];   // Stores a string value
int iInt;            // Stores an integer value
double dDouble;      // Stores a double value

// Loop through all components
for (int iCount = 0; iCount < message.iGetNumComponents(); iCount++)
{
   // Retrieve the components label
   cout << "Message Label: " << message.sGetLabelByNumber(iCount)
        << " " << flush;

   // Retrieve the data type of the component and based on its type store and
   // display the component's value
   message.iGetTypeByNumber(cType, iCount);
   switch(cType)
   {
      case STRING:
         message.iGetValueByNumber(iCount, sString);
         cout << "String Value: " << sString << endl;
         break;
      case INT:
         message.iGetValueByNumber(iCount, &iInt);
         cout << "Int Value: " << iInt << endl;
         break;
      case DOUBLE:
         message .iGetValueByNumber(iCount, &dDouble);
         cout << "Double Value: " << dDouble << endl;
         break;
      default:
         cout << "Unknown Type"" << endl;
         break;
   }
}
```

If the entire XPCComponent element is needed, it can also be retrieved. The method used is GetComponent() (Listing 8.10), which returns a pointer to the XPCComponent object given its ID. If the element specified does not exist, GetComponent() returns NULL, indicating failure.

Listing 8.10 The Overloaded GetComponent() Method Using the ID

```
XPCComponent *GetComponent(int _iID)
{
   int iNum;

   // If the operating system used to retrieve the XPCComponent value differs
   // from the operating system used to store its value, the "endian" value
   // of the data member is returned
   if (cOpSystem != _cOpSystem)
      iNum = iNumComponents.getEndianValue();
   else
      iNum = iNumComponents.getOriginalValue();

   // If the ID specified does not exist, 0 is returned
   if ((_iID < 0) || (_iID >= iNum))
      return (XPCComponent *)NULL;

   return ComponentList[_iID];
}
```

The other way to extract a body component value is by label. Each body component, when inserted, is associated with a label, which can be used to extract the associated value. Accessing body component values by label has the advantage of retrieving values regardless of the ordering of body components. If body components are added or removed or the order in which they are sent is changed, the subscriber does not need to be modified. The subscriber can still access the values needed by retrieving the components by name.

When retrieving a body component value by label, the iGetValueByName() overloaded methods are used. These methods store the value of the body component within a passed parameter. The data type of the parameter dictates the format of the returned variable. If the cType data member of XPCComponent does not match the data type of the parameter, iGetValueByName() returns 0, indicating an error has occurred. If the label requested does not exist, 0 is returned. The value specified by XPCComponent is successfully extracted if the label exists and the parameter's data type matches the data type when the value was originally stored in XPCComponent. Only one iGetValueByName() overloaded method is shown (Listing 8.11) because the rest are similar (see the CD-ROM).

Listing 8.11 The `iGetValueByName()` Method

```
int iGetValueByName(char *_sLabel, char *_sValue, char _cOpSystem)
{
   int iNum;

   // If the operating system used to retrieve the XPCComponent value differs
   // from the operating system used to store its value, the "endian" value
   // of the data member is returned
   if (cOpSystem != _cOpSystem)
      iNum = iNumComponents.getEndianValue();
   else
      iNum = iNumComponents.getOriginalValue();

   // All body components are looped through comparing the label passed in to
   // the XPCComponent's label. If the label is found, its results are stored
   // in the _sValue parameter if the parameter's type matches that of
   // XPCComponent. If the types match 1 is returned otherwise 0 is returned.
   for (int iCount = 0; iCount < iNum; iCount++)
   {
      if (strcmp(ComponentList[iCount]->sGetLabel(), _sLabel) == 0)
         return ComponentList[iCount]->iGetValue(_sValue, _cOpSystem);
   }

   // If the label does not match any of the stored XPCComponent objects, 0 is
   // returned.
   return 0;
}
```

If the original type of the specified `XPCComponent` object is unknown, it can be extracted using the `iGetTypeByName()` method (Listing 8.12). The `XPCComponent` type is returned based on the label specified. If an `XPCComponent` with the specified label does not exist, `iGetTypeByName()` returns `0`, indicating failure; otherwise, `1` is returned and the type is stored within the `_cType` parameter.

Listing 8.12 The iGetTypeByName() Method

```
int iGetTypeByName(char &_cType, char *_sLabel, char _cOpSystem)
{
   int iNum;

   // If the operating system used to retrieve the XPCComponent value differs
   // from the operating system used to store its value, the "endian" value
   // of the data member is returned
   if (cOpSystem != _cOpSystem)
      iNum = iNumComponents.getEndianValue();
   else
      iNum = iNumComponents.getOriginalValue();

   // Array of XPCComponent objects is looped through comparing the label with
   // the one passed in. If the body component's label matches, its type is
   // stored within the _cType parameter and 1 is returned indicating success.
   for (int iCount = 0; iCount < iNum; iCount++)
   {
      if (strcmp(ComponentList[iCount]->sGetLabel(), _sLabel) == 0)
      {
         _cType = ComponentList[iCount]->cGetType();
         return 1;
      }
   }

   // If the label is not found a 0 is returned indicating failure.
   return 0;
}
```

If the entire `XPCComponent` element is needed, it can also be retrieved using the associated label. The method used is `GetComponent()`, which returns a pointer to the `XPCComponent` object given its label. If the element specified does not exist, `GetComponent()` returns `NULL`, indicating failure. The overloaded `GetComponent()` method used to retrieve the `XPCComponent` element by name is shown in Listing 8.13.

Listing 8.13 The Overloaded `GetComponent()` Method Using the Label

```
XPCComponent *GetComponent(char *_sLabel)
{
int iNum;

   // If the operating system used to retrieve the XPCComponent value differs
   // from the operating system used to store its value, the "endian" value
   // of the data member is returned
   if (cOpSystem != _cOpSystem)
      iNum = iNumComponents.getEndianValue();
   else
      iNum = iNumComponents.getOriginalValue();

   // The array of XPCComponent objects is loop through comparing their label
   // with the one passed in. If the body component's label matches, a pointer
   // to the XPCComponent object is returned
   for (int iCount = 0; iCount < iNum; iCount++)
   {
      if (strcmp(ComponentList[iCount]->sGetLabel(), _sLabel) == 0)
         return ComponentList[_ID];
   }

   return (XPCComponent *)NULL;
}
```

Publishing, Subscribing, and Receiving `XPCMessage` Objects

An `XPCMessage` object is published using the `vPublish()` method (Listing 8.14). The components are sent one at a time because

- each component is a fixed size, which allows the receiving end to know the amount of data to store;
- the components are relatively small, and sending them one at a time decreases the chance of overloading the network; and
- the size of `XPCMessage` can grow very large without fear of bogging down the network when transmitting.

First the `XPCHeader` object is sent. This component contains the subject of the message and the number of `XPCComponent` objects that follow. The `XPCTcpSocket` data member is used to transmit the components.

Listing 8.14 The vPublish() Method

```
void vPublish()
{
   int iNum;

   // The number of body components are extracted and stored.
   iNum = iNumComponents.getValue();

   // The header component is sent first
   Socket->iSendMessage((void *)&Header, sizeof(XPCHeader));

   // All body components are transferred next
   for (int iCount = 0; iCount < iNum; iCount++)
      Socket->sendMessage((void *)ComponentList[iCount], sizeof(XPCContainer));
}
```

When an application subscribes to a message, it uses the vSubscribe() method (Listing 8.15) of XPCMessage, which sends the XPCHeader object to inform the network that it is subscribing to a message.

Listing 8.15 The vSubscribe() Method

```
void vSubscribe()
{
   // The message header is sent to notify that the current application is
   // subscribing to the message name contained within the XPCHeader object
   Socket->iSendMessage((void *)&Header, sizeof(XPCHeader));
}
```

After the application subscribes to all messages that it is interested in, it calls vGetMessages() (Listing 8.16) to receive all published messages. Only the messages subscribed to will be received. Because vGetMessages() receives all message publications regardless of their message name and not tied to one particular XPCMessage object, it is a static method. It receives an XPCHeader object and calls a user-specified function, passing it the name of the message, the number of body components it contains, and a pointer to the XPCTcpSocket object used to retrieve the message. A pointer to the user-defined function is passed into vGetMessages() along with the pointer to the XPCTcpSocket object, an XPCHeader object, and the operating system. If multiple subscribed messages are received, vGetMessages() is called within a loop.

Listing 8.16 The `vGetMessages()` Method

```
void vGetMessages (XPCTcpSocket *_Socket,
                   XPCHeader &Header,
                   void vCallBack(XPCTcpSocket *_Socket,
                                  int iNumComponents,
                                  char *_sSubscription),
                   char _cOpSystem = OPSYSTEM)
{
   // The message header is received. The MSG_WAITALL option is specified to
   // make sure that the entire XPCHeader object is received.
   Socket->iRecieveMessage((void *)&Header, sizeof(XPCHeader), MSG_WAITALL);

   // The user-defined function is called and passed a pointer to the
   // XPCTcpSocket object, the number of XPCComponents within the message,
   // and the message name
   vCallBack(_Socket, _Header.iGetCount(_cOpSystem),
             _Header.sGetSubscription());
}
```

8.4 The Publish-and-Subscribe Server

One of the advantages of a publish-and-subscribe system is that it can guarantee message delivery in which clients and servers start and stop as needed. Message delivery is guaranteed when using TCP sockets, but TCP sockets require that the communicating processes be connected. A loss of connection could cause the entire system to fail if not handled properly. If a client disconnects from a server, the disconnection can be detected and handled properly by checking the global error code. In the case of a UNIX system, this error code is stored in the `errno` global variable, and its value is defined by `ECONNRESET`. Windows NT also stores the client's disconnect in its global error variable. This error is defined by `WECONNRESET` and is retrieved from `WSAGetLastError()`. If the server dies and disconnects from the client, the client can no longer communicate and the entire system fails.

UDP sockets do not have the problems that the connected entities face. Processes communicating via UDP sockets are not connected and therefore do not fail if all communicating entities are not up and running at the same time. UDP sockets allow client and server processes to come and go without ever having to check whether all needed processes are active. Unfortunately, the UDP protocol does not guarantee message delivery, and it does not guarantee that the order in which messages are sent are in the same order they are received.

In order to provide the reliability of TCP sockets with the flexibility of the UDP protocol, the publish-and-subscribe system incorporates a server called `PubSubServer`, which is a daemon that runs in the background and connects publishers with subscribers. `PubSubServer` communicates using TCP sockets to ensure message delivery and ordering. Whenever a process publishes or subscribes to a message, it communicates directly with `PubSubServer`. As long as `PubSubServer` is active, publishing and subscribing processes can start and stop as

necessary. The communicating processes do not need knowledge of the destination processes because the publishers and subscribers communicate directly with PubSubServer. Systems described in previous chapters require that the host on which the process is running and the socket port on which the process is communicating be known at run time. Using PubSubServer as an intermediary process, this information is not needed. Only the host name and socket port of PubSubServer is necessary.

Publish-and-subscribe processes never have to create XPCTcpSocket objects that accept connecting clients. Instead, theses processes create XPCTcpSocket objects that connect directly with PubSubServer. Once a connection is established, two-way communication with PubSubServer can take place.

When a process connects, PubSubServer spawns a thread dedicated to publisher and subscriber communication. When a connected process sends a message, it sends an XPCHeader object first. The message type contained within the object is extracted to determine how it should be treated. This message type can be either PUBLISH or SUBSCRIBE. If the message type is SUBSCRIBE, the subscribed message is extracted and, along with the TCP socket of the subscribing process, is added to the PubSubServer subscription list. This subscription list is stored as a linked list of XPCSubscription objects (Listing 8.17).

Listing 8.17 The XPCSubscription Class Definition

```
#include <XPCTcpSocket.h>   // Defines the XPCTcpSocket object

class XPCSubscription
{
   char sSubscribe[1024];   // Message being subscribed
   XPCTcpSocket *Socket;    // TCP socket object that is connected to the
                            // subscriber
public:
   XPCSubscription *next;  // Pointer to the next element in the
                           // XPCSubscription linked-list Constructor.
                           // Stores the message subscribe to and the
                           // socket connected to the subscriber
   XPCSubscription(char *_sSubscribe, XPCTcpSocket *_Socket)
   {
      strcpy(sSubscribe, _sSubscribe);
      Socket = _Socket;
      next = NULL;
   }

   // Returns the message subscription
   char *sGetSubscribe() { return sSubscribe; };
```

```
    // Returns the socket connected to the subscriber
    XPCTcpSocket *getSocket() { return Socket; }
};
```

When a client disconnects from PubSubServer, its XPCTcpSocket object is searched for within the linked list. If it is found, the subscriber is assumed to have died, and its entry is removed from the linked list. Because a process can subscribe to multiple messages, all XPCSubscription objects associated with the subscriber are removed.

When a processes sends an XPCHeader object with the message type PUBLISH, the name of the message and the number of body components being published are extracted. An XPCMessage object is constructed using the number of body components contained within XPCHeader, and the XPCTcpSocket object is used to communicate with the publisher. Constructing an XPCMessage object in this manner builds the XPCMessage object's ComponentList using the XPCComponent object transmitted over the network. Any problem that occurs transmitting the message from the publisher to PubSubServer is caught at this point. During construction of the XPCMessage object, each incoming XPCComponent sequence ID is checked to make sure that all components are received in the order sent. If a discrepancy occurs, the entire message is discarded. If the XPCMessage object is successfully created, the linked list of XPCSubscription objects is traversed, matching the published message name with the subscribed name. The XPCMessage object is sent to all subscribed processes using the XPCTcpSocket object that is stored within XPCSubscription objects and that matches the subscription name.

Each time the XPCSubscription linked list is accessed for traversal, insertion, or deletion, a semaphore is locked to prevent other threads from changing the linked list at the same time another thread is accessing it. The semaphore and the XPCTcpSocket object are passed to the communicating thread and encapsulated within an XPCClientInfo object (Listing 8.18).

Listing 8.18 The XPCClientInfo Class Definition

```
// The Semaphore class defined depends upon the operating system used.
#include <XPCSemaphore.h>

#include <XPCTcpSocket.h>   // Defines the XPCTcpSocket

class XPCConnectInfo
{
   XPCTcpSocket *Socket;   // Pointer to the XPCTcpSocket object used for
                           // receiving messages from the publisher and
                           // subscriber
   XPCSemaphore *Sem;      // Stores the semaphore used to protect the linked
                           // list of connected subscribers
public:
   // Constructor. Stores the pointer to the XPCTcpSocket object as well as
```

```
    // the pointer to the semaphore object
    XPCConnectInfo (XPCTcpSocket *_Socket,
                    XPCSemaphore *_Sem)
            { Socket = _Socket; Sem = _Sem; }

// Member functions that return the private member values
    XPCTcpSocket *GetSocket() { return Socket; }
    XPCSemaphore *GetSem() { return Sem; }
};
```

The state diagram of PubSubServer is shown in Figure 8.5, and a full listing is shown in Listing 8.19. Code specific to an operating system is encapsulated within #ifdef ... #endif code blocks. The PubSubServer shown here communicates on socket port 6800 and can run under the UNIX or Windows NT operating systems.

8.5 Creating a Publish-and-Subscribe System

A publish-and-subscribe system has no server applications — only clients, who can either send messages (publish), receive messages (subscribe), or both. The clients do not communicate directly with each other; rather, they communicate indirectly through PubSubServer. When a client is activated, it creates an XPCTcpSocket object using socket port number 6800 and connects to PubSubServer using the name of the host on which it is executing.

When the client communicates, it sends messages directly to PubSubServer. The message is encapsulated within an XPCMessage object and can be a SUBSCRIBE message or a PUBLISH message, which contains an XPCHeader object with the name of the message and zero or more XPCBodyComponent objects that contain data associated with the message. When PubSubServer receives a published message, it matches the message name to those processes that subscribed to it and forwards the received XPCMessage object.

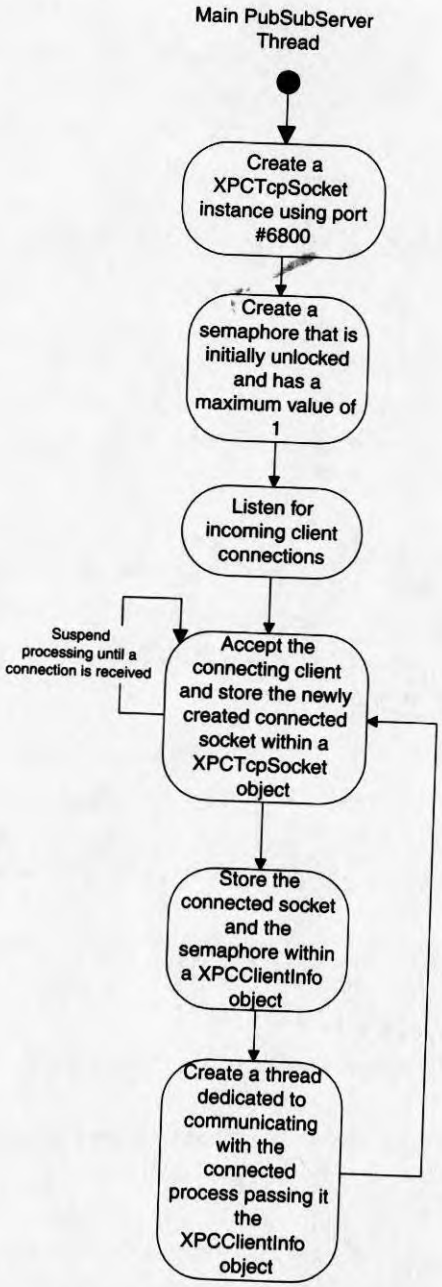
In order for a process to receive subscribed messages, it must use the static vGetMessages() method, which receives the XPCHeader object sent by PubSubServer and calls a user-defined function to process the remainder of the message. The user-defined function can process the message differently based on the subject of the message contained within XPCHeader. Methods for publishing and subscribing to messages are discussed in this section, and UNIX and Windows NT examples are shown.

Overview of the Sample System

The sample system created for this section is a real-time news publication and subscription service. The goal of this system is to simulate news items published from various sources and to retrieve the published news item in real time. The two sources used within this system are "Business News" and "Tech News." These services can start and stop without affecting the other publishers and subscribers within the network. The services publish the following information.

- the actual news item
- the source of the news item
- the priority of the news item

Figure 8.5 **The `PubSubServer` State Diagram**



Listing 8.19 PubSubServer

```
#include <iostream.h>
#include <XPCConnectInfo.h>          // Defines the XPCConnectInfo class
#include <XPCClientConnection.h>     // Defines the XPCClientConnection class
#include <XPCMsg.h>                  // Defines the XPCComponent class as well
                                     // as the XPCMessage

// Operating system specific definition of the thread class
#include <XPCPthread.h>

XPCClientConnection *firstClient;    // Points to the first element of the
                                     // XPCClientConnection linked list

void vDeleteConnection(XPCConnectInfo *_connectInfo)
{
   // vDeleteConnection() deletes all nodes from the linked list that match
   // the socket file descriptor contained within the _connectInfo parameter.
   // Before any linked list nodes are removed, the semaphore is locked
   // to prevent other threads from accessing it while nodes are being removed.
   // The semaphore is unlocked when the linked list traversal is complete.

   XPCClientConnection *currentClient;   // Pointer to current node within the
                                         // linked list

   _connectInfo->getSem()->vLockWait();   // The semaphore is locked

   // Both the current linked list node and the previous linked list node are
   // kept track of
   currentClient = firstClient;
   XPCClientConnection *previousClient = firstClient;

   while(currentClient != NULL)   // While there are still nodes within the
                                  // linked list
   {
      // If the connected socket file descriptor matches the file descriptor
      // within the linked list, the node is removed
      if (currentClient->getSocket()->iGetSocketFd() ==
                              _connectInfo->getSocket()->iGetSocketFd())
      {
```

```
            cout << "Found subscribed socket and removing" << endl;
            if (currentClient == firstClient)
            {
               cout << "Removing ID in first"
                    << _connectInfo->getSocket()->iGetSocketFd() << endl;

               firstClient = firstClient->next;
               currentClient = firstClient;
               previousClient = currentClient;
            }
            else
            {
               previousClient->next = currentClient->next;
               currentClient = previousClient->next;
            }
         }
         else
         {
            previousClient = currentClient;
            currentClient = currentClient->next;
         }
      }

      // The semaphore is unlocked
      _connectInfo->getSem()->vUnlock();
   }

   #ifdef UNIX
   void *vProcessConnect(void *_vArg)
   #else
   DWORD WINAPI vProcessConnect(void *_vArg)
   #endif
   {
      // Client communicating thread. Client can send one of two types of messages:
      // a SUBSCRIBE or a PUBLISH. If it is a SUBSCRIBE message, the message
      // subscription name is added to a linked list of subscriptions. If the
      // message is a PUBLISH message, the linked list of subscriptions is
      // traversed and the message is PUBLISHED to all clients within
      // the subscription linked-list who subscribe to the published message
      XPCConnectInfo *connectInfo = (XPCConnectInfo *)_vArg;
```

```
XPCHeader Header;    // Stores the message header.
int iRet;            // Stores return values from XPCTcpSocket socket
                     // reads

// Points to the current element in the linked-list of subscribed clients
XPCClientConnection *currentClient;

try
{
   while(1)   // Loop forever
   {
      // The XPCHeader object is the first object sent by the connected
      // clients and so is the first message received by the
      // PubSubServer.  The MSG_WAITALL option is sent to ensure that the
      // entire XPCHeader object is received before proceeding
      iRet = connectInfo->getSocket()->iRecieveMessage((void *)&Header,
                                          sizeof(XPCHeader), MSG_WAITALL);

      // If a 0 is returned from iReceiveMessage, the client has
      // disconnected.  All client related subscription linked-list nodes
      // are removed and the thread exits
      if ((iRet == 0)
      {
         vDeleteConnection(connectInfo);
         delete connectInfo;
         return 1;
      }

       if (Header.cGetType() == SUBSCRIBE)
       {
          // The message type is SUBSCRIBE.  The subscription is
          // removed from the XPCHeader object and it along with the
          // connected XPCTcpSocket object are encapsulated within
          // a XPCClientConnection object and added to the linked list

          // The semaphore protecting the linked list is locked
          connectInfo->getSem()->vLockWait();
```

```
        if (firstClient == NULL)
        {
            firstClient = new XPCClientConnection(
                  aPubSubMsg.sGetSubscription(),connectInfo->getSocket());
            currentClient = firstClient;
            currentClient->next = NULL;
        }
        else
        {
            currentClient = firstClient;
            while (currentClient->next != NULL)
               currentClient = currentClient->next;

            XPCClientConnection *newClient = new
                      XPCClientConnection(aPubSubMsg.sGetSubscription(),
                                          connectInfo->getSocket());
            currentClient->next = newClient;
            currentClient = currentClient->next;
            currentClient->next = NULL;
        }

        // The semaphore protecting the linked-list is unlocked
        connectInfo->getSem()->vUnlock();
    }
    else if (Header.cGetType() == PUBLISH)
    {
        // If the message type is PUBLISH. A XPCMessage object
        // is constructed to store the received Header and the
        // incoming XPCComponent objects. When the XPCMessage
        // object is fully constructed, the linked list of
        // subscribed clients are parsed. The XPCMessage object
        // is published to all clients within the linked list who've
        // subscribed to the published message.

        // XPCMessage is constructed and receives all incoming
        // XPCComponent objects
        XPCMessage newMessage(aPubSubMsg.iGetCount(),
                              connectInfo->getSocket());
```

```
                // The linked-list of subscribed clients is traversed and
                // the XPCMessage object is sent to all clients whose
                // subscription matches the published message.
                currentClient = firstClient;
                while(currentClient != NULL)
                {
                   if (strcmp(currentClient
                        ->sGetSubscribe(), aPubSubMsg.sGetSubscription()) == 0)
                   {
                        currentClient->getSocket()->sendMessage(
                                    (void *)&aPubSubMsg, sizeof(XPCPubSubMsg));
                        for (int iCount = 0; iCount < aPubSubMsg.iGetCount();
                             iCount++)
                            currentClient->getSocket()->sendMessage(
                                    (void *)newMessage.ContainerList[iCount],
                                    sizeof(XPCContainer));
                   }
                   currentClient = currentClient->next;
                }
             }
          }
       }
       catch(XPCException &exceptOb)
       {
          // All exceptions are caught and displayed to the user
          cout << "Client communication error: " << exceptOb.sGetException()
               << endl;
          vDeleteConnection(connectInfo);
          delete connectInfo;
          return 1;
       }
}

main()
{
   // Definition of the semaphore used to protect access to the linked list
   // of XPCClientConnection objects. The semaphore is initialized to
   // unlocked with only values of unlocked and locked allowed.
#ifdef UNIX
```

```
    XPCSemaphore listSem(PRIVATE_SEM, 1, 1);
#else
    XPCSemaphore listSem("LIST SEM", 1, 1);
#endif
    try
    {
        // A socket is setup using port number 6800 and bound to
        XPCTcpSocket serverSocket((long int)6800);
        serverSocket.vBindSocket();
        serverSocket.vListen(5);

        while(1)   // Loop forever
        {
            // Incoming client connections are accepted
            XPCTcpSocket *newConnection = serverSocket.vAccept();

            // A XPCConnectInfo object is constructed using the newly created
            // XPCTcpSocket object and the semaphore
            XPCConnectInfo *connectInfo
                                = new XPCConnectInfo(newConnection, &listSem);
            // A thread is spawned and passed a pointer to the XPCConnectInfo
            // object. The thread is dedicated to communicating with the
            // connected client
#ifdef UNIX
            XPCPthread<int> connectThread(vProcessConnect, (void *)connectInfo);
#else
            XPCThread connectThread(processConnect, (void *)connectInfo)
#endif
        }
    }
    catch(XPCException &exceptObject)
    {
        cout << "Socket Error: " << exceptObject.sGetException() << endl;
    }
}
```

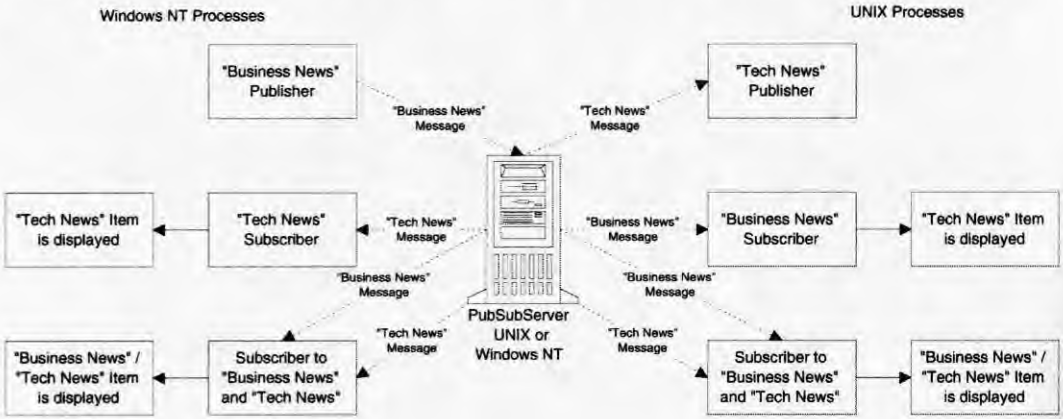
The news item and news item source are strings, whereas the priority indicator is an integer. Processes subscribe to one or both of the news sources and receive all published items. Like publishers, subscribers can come and go without affecting the other processes within the system. When a subscriber receives a message, it displays the message to the user in the following format.

<Priority Level>: From <Service> Issued by <Source> - <News Item>

The <Priority Level> can be Critical, Important, or Informative. <Service> is either Business News or Tech News, depending on the source from which the message is received. <Source> stands for the service from which the news item was received. The news item source is included within the message received. <News Item> is the actual news being published.

Both the publishers and subscribers are developed on UNIX and Windows NT. The Windows NT applications are developed as both console and Windows executables. The `PubSubServer` being used can run on either UNIX or Windows NT since both versions serve the same purpose. Regardless of the operating systems, the components that compose the news publication and subscription service are the same. The basic news publication and subscription service system is shown in Figure 8.6.

Figure 8.6 The News Publication and Subscription System



The News Publishers

The News Publisher is the simplest of the real-time news publications. Each news publication process is predefined to publish one type of message: Business News or Tech News. At application initialized, the user is prompted for the news message, the source of the news item, and a priority indicator.

When the publisher initializes, it creates an `XPCTcpSocket` object constructed with socket port number 6800. Upon successful construction of the `XPCTcpSocket` object, it is connected to `PubSubServer` using its `vConnect()` method, passing `vConnect()` the name of the host on which `PubSubServer` is executing.

When the publisher is connected to `PubSubServer`, the user is prompted for information pertaining to the news item. The user is required to input the news item, the source from which

the news originated, and the priority level in the order given. If the user enters QUIT in place of the news item, the News Publisher exits; otherwise, an XPCMessage object is constructed with the subject "Business News" or "Tech News," depending on the type of publisher. Three XPCBodyComponent objects are specified upon construction, and a pointer to the connected XPCTcpSocket object is passed to the constructor. For each piece of the news message, an XPCBodyComponent object representing the user's entry is inserted into XPCMessage by using the vInsert() method and assigning the component value a label. When all three XPCBodyComponent objects are inserted, the XPCMessage object is published using the vPublish() method.

The first News Publisher process presented is a UNIX/Windows NT console application (Listing 8.20). Each application is assigned a specific service: BUSINESS or TECH. The example presented publishes BUSINESS news items. The console application that publishes TECH messages is on the CD-ROM. The following code can be compiled on either UNIX or Windows NT.

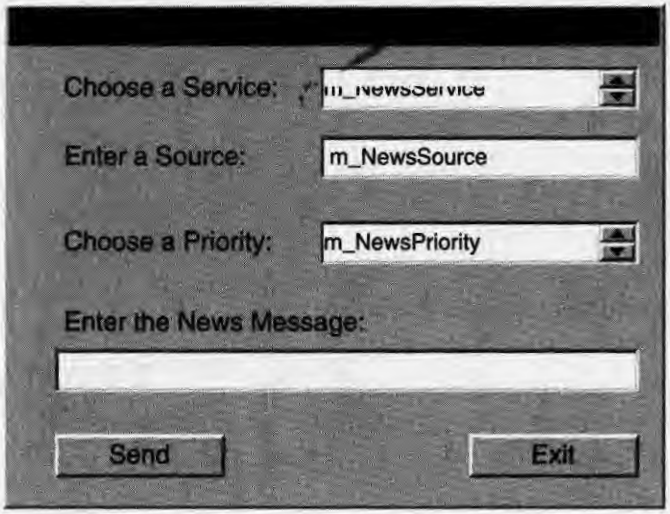
Listing 8.20 The Console-Based News Publisher

```
#include <XPCMsg.h>          // Defines the XPCMsg object
#include <iostream.h>

main(int argc, char *argv[])
{
   char sBuf[256];      // Stores the user-entered values

   try
   {

      // A XPCTcpSocket object is created using socket port #6800 and
      // connected to the host specified on the command-line. The host
      // specified must be executing the PubSubServer application
      XPCTcpSocket businessSocket((long int)6800);
      businessSocket.vConnect(argv[1]);

      while(1)   // Loop forever
      {
         // The user is prompted for the news message. If the user enters
         // "QUIT", the client exits.
         cout << "Enter News Item: " << flush;
         cin.getline(sBuf, 256);

         if (strcmp(sBuf, "QUIT") == 0)
            break;
```

```
        // A new XPCMessage object is created. Three XPCBodyComponent
        // objects are specified and the message is given the name of
        // "BUSINESS".
        XPCMessage newMessage(3, "BUSINESS", PUBLISH, &businessSocket);

        // The user-entered news item is inserted into the XPCMessage
        // object and is given a label of "News"
        newMessage.vInsert("News", sBuf);

        // The source of the news item is prompted for.
        cout << "Enter Source: " << flush;
        cin.getline(sBuf, 256);

        // The user-entered source is inserted into the XPCMessage
        // object and is given a label of "Source"
        newMessage.vInsert("Source", sBuf);

        // The priority level of the news item is prompted for.
        cout << "Enter Priority Level: " << flush;
        cin.getline(sBuf, 256);

        // The user-entered priority is converted to an integer and
        // inserted into the XPCMessage object. The priority is given a
        // label of "Priority"
        newMessage.vInsert("Priority", atoi(sBuf));

        // The XPCMessage object is published
        newMessage.vPublish();
    }
  }
  catch(XPCException &exceptOb)
  {
    // All socket communication related exceptions are caught and displayed
    // to the user.
    cout << "Communication Error: " << exceptOb.sGetException() << endl;
      return 0;
  }

  return 1;
}
```

It might be preferable to develop the publishing application using a GUI interface on Windows NT as a dialog-based application. An MFC dialog application is encapsulated within a class that inherits from the `CDialog` class. The dialog used for the News Publisher allows the user to choose the type of new service, the source of the news item, the priority for the news item, and the news message itself. The News Publisher dialog is displayed in Figure 8.7.

Figure 8.7 The News Publisher Dialog



The Windows component objects are encapsulated within the main News Publisher dialog class. This class is referred to as `CNewsPublisherDlg`, and an instance of this object is created when the dialog application executes. Because multiple methods of `CNewsPublisherDlg` can create and publish messages, the `XPCTcpSocket` object is a private data member of `CNewsPublisherDlg` (Listing 8.21).

Listing 8.21 The `CNewsPublisherDlg` Class

```
#include <XPCMsg.h>       // Defines the XPCMsg class
#include <iostream.h>

class CNewsPublisherDlg : public CDialog
{
   XPCTcpSocket *clientSocket;   // Socket used to communicate to the
// PubSubServer
public:
   CNewsPublisherDlg(CWnd* pParent = NULL);   // standard constructor
```

```
// Dialog Data
   //{{AFX_DATA(CNewsPublisherDlg)
   enum { IDD = IDD_NEWSPUBLISHER_DIALOG };
   CEdit      m_NewsMessage;      // Contains the user-entered news message
   CEdit      m_NewsSource;       // Contains the user-entered news source
   CComboBox   m_NewsPriority;    // Contains a list of possible priorities
   CComboBox   m_NewsService;     // Contains a list of news services
   //}}AFX_DATA

   ...

// Implementation
protected:
   HICON m_hIcon;

   // Generated message map functions
   //{{AFX_MSG(CNewsPublisherDlg)
   virtual BOOL OnInitDialog();    // Called when the News Publishing dialog
                                   // initializes. Does system setup prior to
                                   // the dialog's display
   ...
   afx_msg void OnExitButton();      // Called when the user clicks the "Exit"
                                     // button
   afx_msg void OnSendButton();      // Called when the user clicks the "Send"
                                     // button
   //}}AFX_MSG
   DECLARE_MESSAGE_MAP()
};
```

When the Windows application executes, the `OnInitDialog()` method of the inherited `CDialog` class is called. `OnInitDialog()` performs preparation operations prior to displaying the GUI, inserts entries (`BUSINESS` and `TECH`) into the news services combo box (`m_NewsServiceCombo`), and inserts priority values (Critical, Important, and Informative) into the Priority Level combo box (`m_NewsPriorityCombo`). The `OnInitDialog()` method also creates the `XPCTcpSocket` object using socket port number 6800 and calls its `vConnect()` method to connect to the `PubSubServer` that is executing on the specified host. The `CNewsPublisherDlg` `OnInitDialog()` method is shown in Listing 8.22.

Listing 8.22 The `OnInitDialog()` Method

```
BOOL CNewsPublisherDlg::OnInitDialog()
{
   CDialog::OnInitDialog();

   ...
   // WinSock library is initialized
   AfxSocketInit();

   // The types of news services are added to the news services combo box
   m_NewsService.AddString("BUSINESS");
   m_NewsService.AddString("TECH");
   m_NewsService.SetCurSel(-1);

   // The priority types are added to the priority combo box
   m_NewsPriority.AddString("Critical");
   m_NewsPriority.AddString("Important");
   m_NewsPriority.AddString("Informative");
   m_NewsPriority.SetCurSel(-1);

   try
   {
      // The socket is created using port #6800 and connected to the host
      // executing the PubSubServer
      clientSocket = new XPCTcpSocket((long int)6800);
      clientSocket->vConnect("PubSubServer_Host");
   }
   catch(XPCException &exceptOb)
   {
      // All socket related errors are caught and displayed to the user
      MessageBox(exceptOb.sGetException(), "TCP Error", MB_OK);
      CDialog::OnOK();
   }
   return TRUE;  // return TRUE  unless you set the focus to a control
}
```

A news message is published when the user hits the Send button and calls the `OnSendButton()` method (Listing 8.23) of `CNewsPublisherDlg`. An `XPCMessage` object is constructed using the news service chosen as its message name. The `XPCMessage` `CBodyComponent` objects are inserted using the values entered into the various dialog components. The component

values are extracted and inserted using the XPCMessage vInsert() method. When all values are extracted and inserted into XPCMessage, its vPublish() method is called to send the complete message to the PubSubServer. The remainder of the Windows-based News Publisher dialog application is not shown but can be found on the included CD-ROM.

Listing 8.23 The OnSendButton() Method

```
void CNewsPublisherDlg::OnSendButton()
{
   char sBuf[256];      // Temporary storage for the user-entered data

   try
   {
      // The news service is retrieved
      m_NewsService.GetWindowText(sBuf, sizeof(sBuf));

      // A XPCMessage object is created for publication. The message name
      // give is the news service chosen by the user
      XPCMessage newMessage(3, sBuf, PUBLISH, clientSocket);

      // The news item is extracted from the GUI and inserted into the
      // XPCMessage object
      m_NewsMessage.GetWindowText(sBuf, sizeof(sBuf));
      newMessage.vInsert("News", sBuf);

      // The news source is extracted from the GUI and inserted into the
      // XPCMessage object
      m_NewsSource.GetWindowText(sBuf, sizeof(sBuf));
      newMessage.vInsert("Source", sBuf);

      // The news priority is extracted from the GUI and the corresponding
      // priority value in inserted into the XPCMessage object
      m_NewsPriority.GetWindowText(sBuf, sizeof(sBuf));
      if (strcmp(sBuf, "Critical") == 0)
         newMessage.vInsert("Priority", 1);
      else if (strcmp(sBuf, "Important") == 0)
         newMessage.vInsert("Priority", 2);
      else
         newMessage.vInsert("Priority", 3);
```

```
        // The XPCMessage object is published
        newMessage.vPublish();
    }
    catch(XPCException &exceptOb)
    {
        // All socket related exception are caught and displayed
        MessageBox(exceptOb.sGetException(), "Publish Error", MB_OK);
        return;
    }
}
```

The News Subscribers

The subscribers to news messages listen to one or many news services. The published message has the name of the news service publishing it. The news messages received are formatted and displayed. Many techniques can enhance the ways that published messages are received and processed. These methods include

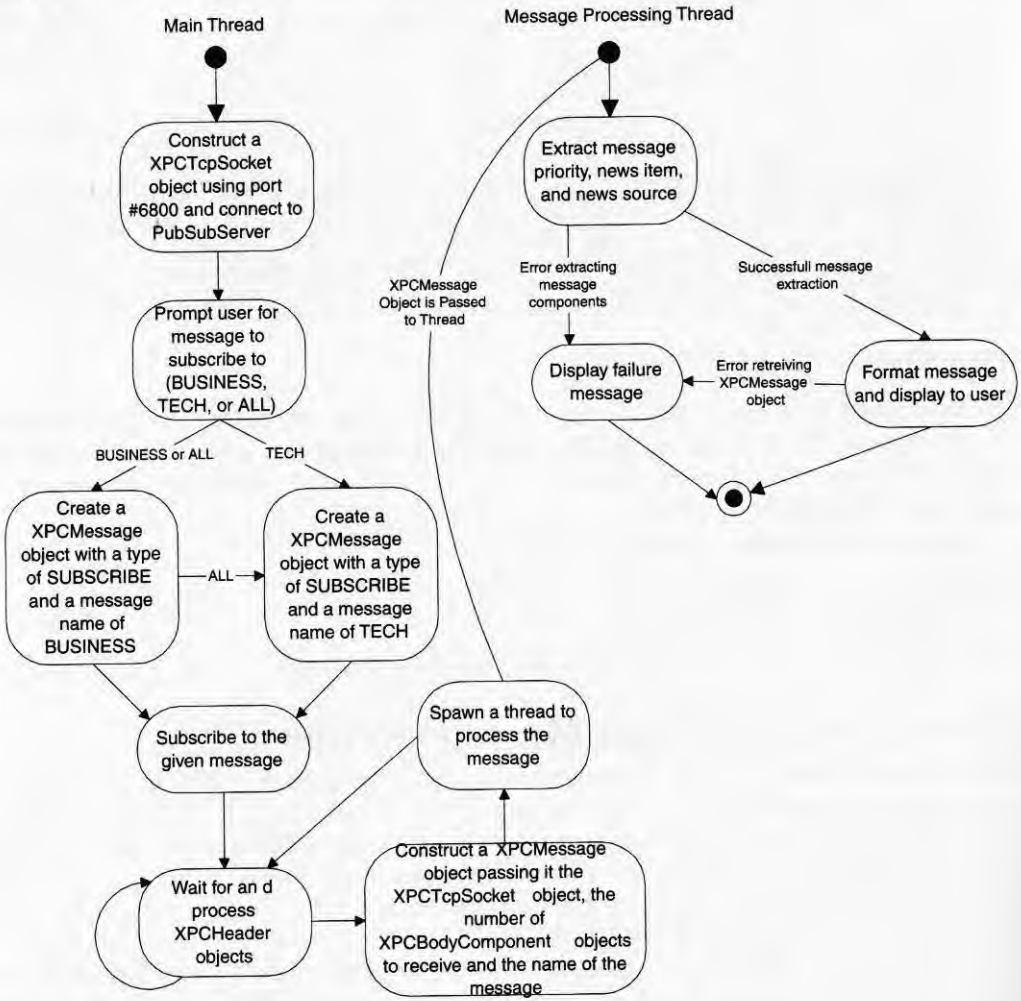
- single-threaded message retrieval,
- multithreaded message retrieval,
- multiple-socket message retrieval, and
- asynchronous message retrieval within a Windows application.

Single-Threaded News Message Retrieval

When the subscriber initializes, it connects to `PubSubServer` and prompts the user to enter the news messages to subscribe to. The user can choose `BUSINESS`, `TECH`, or both by specifying `ALL`. An `XPCMessage` object is constructed for each message subscribed to, and the object is sent to `PubSubServer`. After it has successfully subscribed, the News Subscriber waits to receive published messages. Because it has subscribed to a particular set of messages, `PubSubServer` sends only those messages.

The News Subscriber waits to receive `XPCHeader` objects. It uses the static `vGetMessages()` method of `XPCMessage` and passes an `XPCHeader` object and the `XPCTcpSocket` object used to receive socket messages. `vGetMessages()` suspends processing until an `XPCHeader` object is received. Once the `XPCHeader` object is received, a user-defined callback function processes the incoming message. This callback function is passed as a pointer to `vGetMessages()`. The callback function is passed the `XPCTcpSocket` object used to receive the incoming message, the number of `XPCBodyComponent` objects contained with the published message, and the name of the message being received. The number of `XPCBodyComponent` objects and the name of the received message is extracted from the `XPCHeader` object. The user-defined callback function constructs an `XPCMessage` object using the information passed. `XPCMessage` finishes construction when all `XPCBodyComponent` objects have been received. After `XPCMessage` is received, the data within the object is formatted and displayed to the user. Only after the messages have been fully processed does the callback function return, allowing the receipt of additional news messages. The program flow for the single-threaded News Subscriber is shown in Figure 8.8, and the code for the single-threaded News Subscriber is shown in Listing 8.24.

Figure 8.8 The Single-Threaded News Subscriber



Listing 8.24 The Single-Threaded News Subscriber

```
#include <XPCMsg.h>      // Defines XPCMessage and XPCBodyComponent
#include <Priority.h>    // Assigns definitions to news priority values
#include <iostream.h>

void vProcessNews (XPCTcpSocket *_Socket, int _iNumComponents, char *_sMsgName)
{
   try
   {
```

```
// A XPCMessage object is constructed using message components sent
// over the network
XPCMessage newMessage(_iNumComponents, _Socket, _sMsgName);

char sNewsItem[1024];      // Holds the formatted news message
int iPriority;             // Holds the priority value of the news
                           // message
char sPriority[256];       // Holds the news priority string
                           // representation
char sNews[256];           // Holds the news message
char sSource[256];         // Holds the news source

// The priority of the news item is extracted.
if (!newMessage.iGetValueByName("Priority", &iPriority))
   iPriority = UNKNOWN;

// The news message is extracted
if (!newMessage.iGetValueByName("News", sNews))
   strcpy(sNews, "UNKNOWN");

// The news source is extracted
if (!newMessage.iGetValueByName("Source", sSource))
   strcpy(sSource, "UNKNOWN");

// A string representation of the priority is stored based on the
// priority value extracted from the XPCMessage object
switch(iPriority)
{
   case UNKNOWN:
      strcpy(sPriority, "UNKNOWN");
      break;
   case CRITICAL:
      strcpy(sPriority, "Critical");
      break;
   case IMPORTANT:
      strcpy(sPriority, "Important");
      break;
   case INFORMATIVE:
      strcpy(sPriority, "Informative");
      break;
```

```
    }

    // The entire news message is formatted and displayed to the user.
    sprintf(sNewsItem, "%s : From %s Issued by %s - %s",
            sPriority, _sMsgName, sSource, sNews);
                cout << sNewsItem << endl;
    }
    catch(XPCException &exceptOb)
    {
       // All news processing errors are caught and displayed to the user
       cerr << "Error receiving news message from "
            << _sMsgName << ": " << exceptOb.sGetException() << endl;
    }
}

main(int argc, char *argv[])
{
    char sSubscription[50];    // Stores the user-chosen subscription
    XPCMessage *Subscribe;     // Defines the XPCMessage subscription object
    XPCHeader Header;          // Stores the published XPCHeader object

    // An XPCTcpSocket object is constructed using socket port #6800 and
    // connected to a host executing the PubSubServer
    XPCTcpSocket SubscriptionSocket((long int)6800);
    SubscriptionSocket.vConnect("SubPubServer_Host");

    try
    {
       // The user is prompted for the message subscription.
       cout << "Enter The Message You Wish to Subscribe (BUSINESS, TECH, ALL): "
            << flush;
       cin >> sSubscription;

       // If the name chosen is ALL or BUSINESS, the BUSINESS message is
       // subscribed
       if ((strcmp(sSubscription, "ALL") == 0) ||
                   (strcmp(sSubscription, "BUSINESS") == 0))
       {
          Subscribe = new XPCMessage(0, "BUSINESS", SUBSCRIBE,
                                     &BusSubscriptionSocket);
```

```
            Subscribe->vSubscribe();
            delete Subscribe;
         }

         // If the name chosen is ALL or TECH, the TECH message is subscribed
         if ((strcmp(sSubscription, "ALL") == 0) ||
             ((strcmp(sSubscription, "TECH") == 0)))
         {
            Subscribe = new XPCMessage(0, "TECH", SUBSCRIBE,
                                       &TechSubscriptionSocket);
            Subscribe->vSubscribe();
            delete Subscribe;
         }

         while(1)   // Loop forever
         {
            // Published messages are retrieved using the XPCTcpSocket
            // object.  The XPCHeader object is received and the vProcessNews()
            // function is called to process the message
            XPCMessage::vGetMessages(&SubscriptionSocket, Header, vProcessNews);
         }
      }
      catch(XPCException &exceptOb)
      {
         // All socket communication errors are caught and displayed
         cout << "Communication Error: " << exceptOb.sGetException() << endl;
         return 0;
      }

      return 1;
   }
```

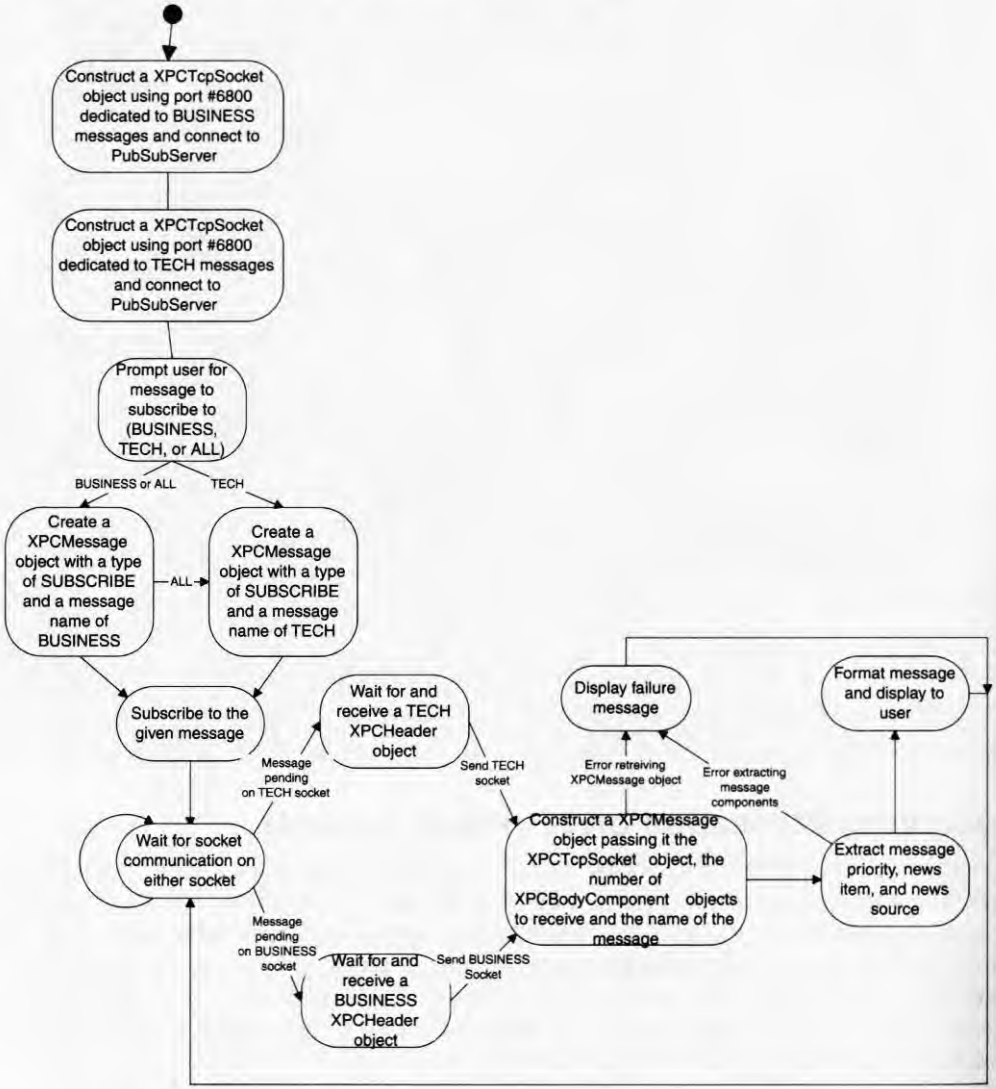
Receiving Messages on Multiple Sockets

Sometimes it is useful to use a specific socket connection to process a specific subscribed message. The approach shown in Listing 8.24 uses a single `XPCTcpSocket` object to subscribe to all chosen messages. Any subscribed messages are sent to the News Subscriber on the socket port originally subscribed to the message. Using separate sockets to receive subscribed messages forces dedicated socket connections. Dedicating socket connections can be useful for determining how to process a message without having to fully examine it. This is possible if specific messages are received on specific sockets.

When receiving messages on multiple socket connections, `select()` is used. It notifies the application on which socket the message is being received. This knowledge can be used to call `vGetMessages()` using the appropriate connected socket. Because the type of message is known, `vGetMessages()` can receive different user-defined message-handling functions customized to the type of message received.

The multisocket news subscription process uses a separate `XPCTcpSocket` object for subscribing and receiving `BUSINESS` and `TECH` messages. The algorithm for the multisocket News Subscriber is shown in Figure 8.9. A partial code listing for the multisocket News Subscriber is shown in Listing 8.25 that details the changes required for communicating on multiple sockets.

Figure 8.9 Multisocket News Subscriber



Listing 8.25 The Multisocket News Subscriber

```
#include <XPCMsg.h>      // Defines the XPCMessage and XPCBodyComponent
#include <Priority.h>    // Assigns definitions to news priority values
#include <iostream.h>

void vProcessNews(XPCTcpSocket *_Socket, int _iNumComponents, char *_sMsgName)
{
   // This function retrieves, processes, and displays an incoming message.
   // See Listing 8.24
}

main(int argc, char *argv[])
{
   char sSubscription[50];   // Stores the user-chosen subscription
   XPCMessage *Subscribe;    // Defines the XPCMessage subscription object
   XPCHeader Header;         // Stores the published XPCHeader object
   int iMaxSocketFd = 0;     // Stores the maximum file descriptor value
   fd_set fdset;             // Stores the set of socket file descriptors

   try
   {
      // A XPCTcpSocket object is created and connected to the PubSubServer.
      // This XPCTcpSocket object is dedicated to subscribing to and
      // receiving BUSINESS messages.
      XPCTcpSocket BusSubscriptionSocket((long int)6800);
      BusSubscriptionSocket.Connect(argv[1]);

      // A XPCTcpSocket object is created and connected to the PubSubServer.
      // This XPCTcpSocket object is dedicated to subscribing to and
      // receiving TECH messages.
      XPCTcpSocket TechSubscriptionSocket((long int)6800);
      TechSubscriptionSocket.Connect(argv[1]);

      // The largest socket file descriptor is chosen and stored
      if (BusSubscriptionSocket.iGetSocketFd() >
          TechSubscriptionSocket.iGetSocketFd())
         iMaxSocketFd = BusSubscriptionSocket.iGetSocketFd() + 1;
      else
         iMaxSocketFd = TechSubscriptionSocket.iGetSocketFd() + 1;
```

```
// User is prompted for the name of the message to which to subscribe
cout << "Enter The Message You Wish to Subscribe (BUSINESS, TECH, ALL): "
     << flush;
cin >> sSubscription;

// If the name chosen is ALL or BUSINESS, the BUSINESS message is
// subscribed
if ((strcmp(sSubscription, "ALL") == 0) ||
    ((strcmp(sSubscription, "BUSINESS") == 0)))
{
   Subscribe = new XPCMessage(0, "BUSINESS", SUBSCRIBE,
                              &BusSubscriptionSocket);
   Subscribe->vSubscribe();
   delete Subscribe;
}

// If the name chosen is ALL or TECH, the TECH message is subscribed
if ((strcmp(sSubscription, "ALL") == 0) ||
    ((strcmp(sSubscription, "TECH") == 0)))
{
   Subscribe = new XPCMessage(0, "TECH", SUBSCRIBE,
                              &TechSubscriptionSocket);
   Subscribe->vSubscribe();
   delete Subscribe;
}

while(1)   // Loop forever
{
   FD_ZERO(&fdset);   // The set of socket file descriptors is
                      // cleared

   // The business and tech socket file descriptors are added to the
   // set
   FD_SET(BusSubscriptionSocket.iGetSocketFd(), &fdset);
   FD_SET(TechSubscriptionSocket.iGetSocketFd(), &fdset);

   // An asynchronous selection of sockets that have incoming
   // message is established
```

```
        int iSelectRetValue = select(iMaxSocketFd, &fdset, NULL, NULL, 0);
        if (iSelectRetValue == -1)
        {
           cerr << "Select failed" << endl;
           return 1;
        }

        // If an incoming message is detected, the correct
        // XPCTcpSocket object is chosen and is used to receive the
        // message.
      for (int iCount = 0; iCount < iSelectRetValue; iCount++)
      {
         if (FD_ISSET(BusSubscriptionSocket.iGetSocketFd(), &fdset) != 0)
         {
            // The message received is on the business related
            // socket.  The XPCHeader object is received and
            // vProcessNews() is called
            XPCMessage::vGetMessages(&BusSubscriptionSocket, Header,
                                     vProcessNews);
         }
         else if (FD_ISSET(TechSubscriptionSocket.iGetSocketFd(),
                           &fdset) != 0)
         {
            // The message received is on the tech related
            // socket.  The XPCHeader object is received and
            // vProcessNews() is called
               XPCMessage::vGetMessages(&TechSubscriptionSocket,
                                        Header,
                                        vProcessNews);
         }
      }
   }
}
catch(XPCException &exceptOb)
{
   // All socket related errors are caught and displayed to the user
   cout << "Communication Error: " << exceptOb.sGetException() << endl;
   return 0;
```

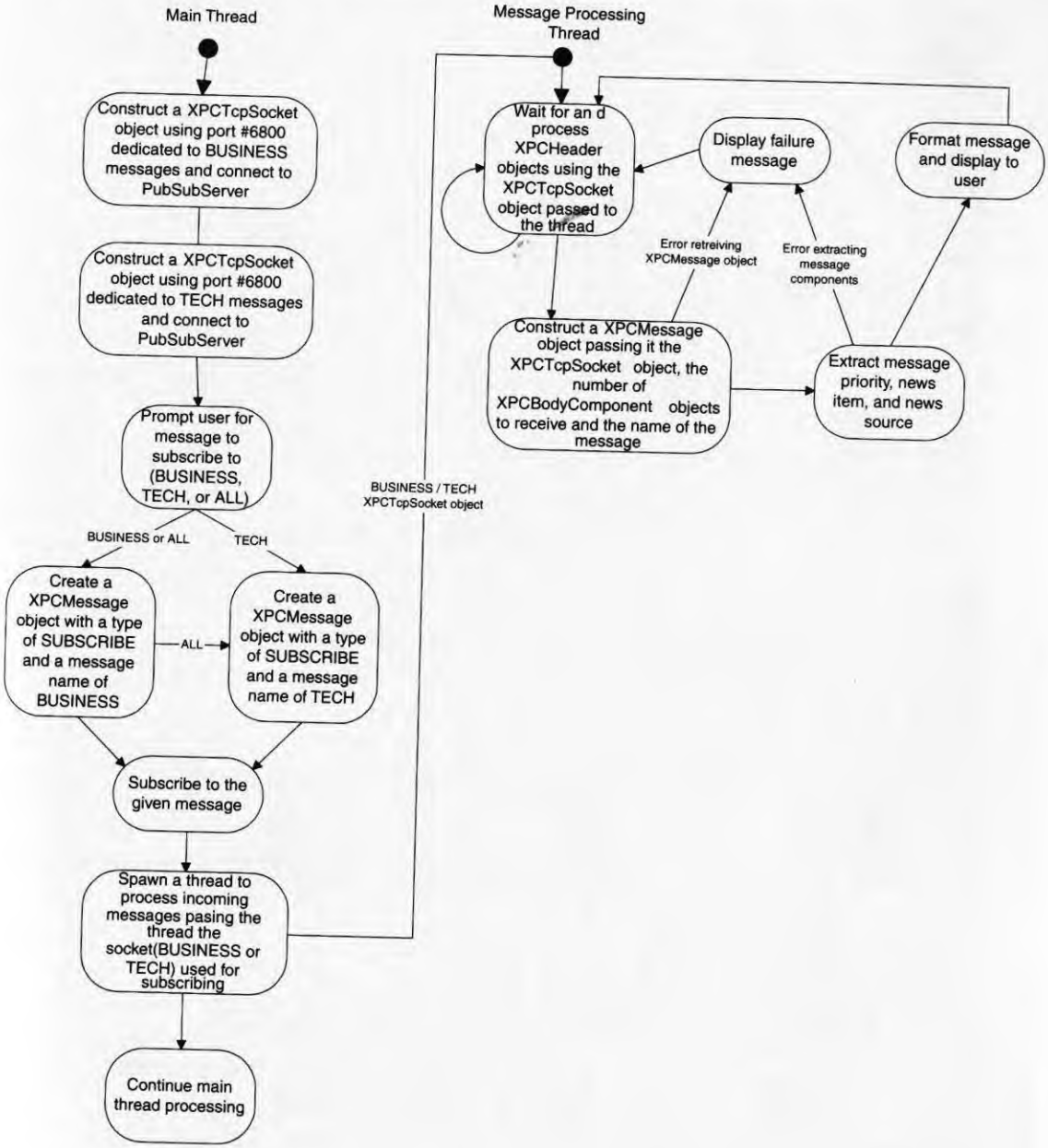
```
    }

    return 1;
}
```

Threading Message Retrieval

A problem common between the two News Subscribers presented is that the main application is suspended regardless of whether or not messages are being received. As soon as the `XPCMessage::vGetMessages()` method is called, the main process does no other processing. One way this situation can be resolved is to thread the `vGetMessages()` process.

After all messages are subscribed, a thread is spawned and passed a pointer to the `XPCTcpSocket` object that communicates with `PubSubServer`. `vGetMessages()` is placed within a loop inside the thread, enabling the main thread to continue processing. Like all threaded programs, it is important that semaphores protect shared global resources. The architecture for the threaded message retrieval News Subscriber is shown in Figure 8.10. The code fragment in Listing 8.26 for the threaded message retrieval News Subscriber details the changes required for communicating on multiple sockets. Code specific to an operating system is encapsulated within `#ifdef ... #endif` code blocks.

Figure 8.10 Threaded Message Retrieval News Subscriber



Listing 8.26 Threaded Message Retrieval News Subscriber

```
#include <XPCMsg.h>      // Defines the XPCMessage and XPCBodyComponent
#include <Priority.h>    // Assigns definitions to news priority values
#include <iostream.h>

// The appropriate thread class is defined
#ifdef UNIX
#include <XPCPthread.h>
#else
#include <Thread.h>
#endif

void vProcessNews(XPCTcpSocket *_Socket, int _iNumComponents, char *_sMsgName)
{
   // Retrieves, processes, and displays an incoming message. See
   // Listing 8.24
}

#ifdef UNIX
void *vSubscribeThread(void *vArg)
#else
DWORD WINAPI vSubscribeThread(void *vArg)
#endif
{
   XPCTcpSocket *Socket = (XPCTcpSocket *)vArg;

   XPCHeader Header;  // Stores the XPCHeader object received from the network

   try
   {
      while(1)   // Loop forever
      {
         // The XPCHeader object from published messages are received using
         // the XPCTcpSocket object passed to the thread. The
         // vProcessNews() function is called to process the remainder of
         // the published message
         XPCMessage::vGetMessages(Socket, Header, vProcessNews);
      }
   }
```

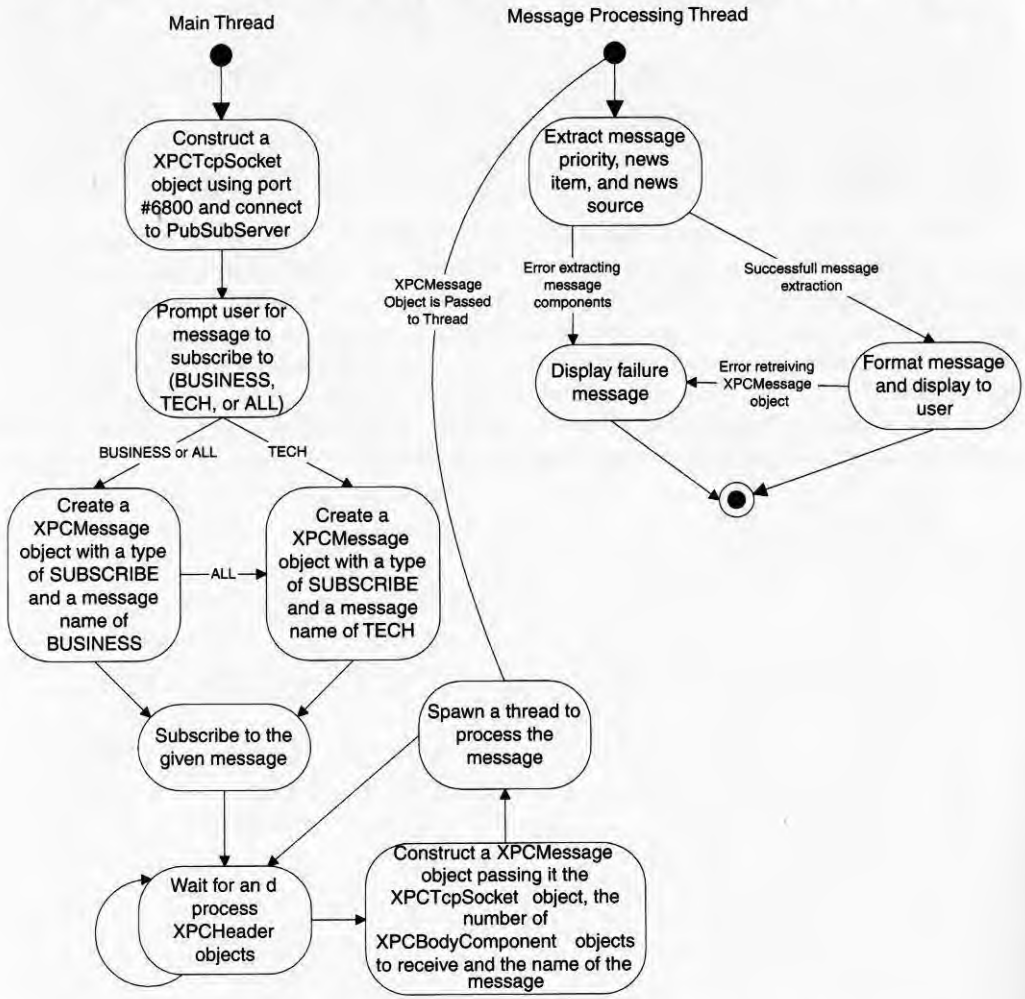
```
    catch(XPCException &exceptOb)
    {
        // All socket communication errors are caught and displayed to the user
        cout << "Communication error: " << exceptOb.sGetException() << endl;
        return 0;
    }

    return 1;
}

main(int argc, char *argv[])
{
    char sSubscription[50];    // Stores the user-chosen subscriptoin
    XPCMessage *Subscribe;     // Defines the XPCMessage subscription object

    try
    {
        // A XPCTcpSocket object is created and connected to the PubSubServer.
        // This XPCTcpSocket object is dedicated to subscribing to and
        // receiving BUSINESS messages.
        XPCTcpSocket BusSubscriptionSocket((long int)6800);
        BusSubscriptionSocket.vConnect(argv[1]);

        // A XPCTcpSocket object is created and connected to the PubSubServer.
        // This XPCTcpSocket object is dedicated to subscribing to and
        // receiving TECH messages.
        XPCTcpSocket TechSubscriptionSocket((long int)6800);
        TechSubscriptionSocket.vConnect(argv[1]);

        // User is prompted for the name of the message to which to subscribe
        cout << "Enter The Message You Wish to Subscribe (BUSINESS, TECH, ALL): "
             << flush;
        cin >> sSubscription;

        // If the name chosen is ALL or BUSINESS, the BUSINESS message is
        // subscribed
        if ((strcmp(sSubscription, "ALL") == 0) ||
            ((strcmp(sSubscription, "BUSINESS") == 0)))
        {
```

```
        Subscribe = new XPCMessage(0, "BUSINESS", SUBSCRIBE,
                                   &BusSubscriptionSocket);
        Subscribe->vSubscribe();
        delete Subscribe;
      }

      // If the name chosen is ALL or TECH, the TECH message is subscribed
      if ((strcmp(sSubscription, "ALL") == 0) ||
          ((strcmp(sSubscription, "TECH") == 0)))
      {
        Subscribe = new XPCMessage(0, "TECH", SUBSCRIBE,
                                   &TechSubscriptionSocket);
        Subscribe->vSubscribe();
        delete Subscribe;
      }

      // Two threads are created. One thread processes BUSINESS messages and
      // the other thread processes TECH messages.
#ifdef UNIX
     XPCPthread<int> BusinessThread(vSubscribeThread,
                                   (void *)&BusSubscriptionSocket);
      XPCPthread<int> TechThread(vSubscribeThread,
                                 (void *)&TechSubscriptionSocket);
#else
      XPCThread BusinessThread(vSubscribeThread,
                               (void *)&BusSubscriptionSocket);
      XPCThread TechThread(vSubscribeThread, (void *)&TechSubscriptionSocket);
#endif

      // The main program thread can perform additional processing if needed

      // The main program thread execution is suspended while the threads
      // execute
      BusinessThread.vWaitForThread();
      TechThread.vWaitForThread();
    }
    catch(XPCException &exceptOb)
    {
      // All socket related communication errors are caught and displayed to
```

```
        // the user
        cout << "Communication Error: " << exceptOb.sGetException() << endl;
        return 0;
    }

    return 1;
}
```

Another approach to multithreading a message subscriber is to thread message processing, which enables the application to receive additional messages while processing previously received messages. Many messages can then be received and processed in parallel, which will improve the performance of the system. The News Subscriber spawns a thread within the message-processing function and passes the same parameters it was passed. The architecture for the threaded message-processing News Subscriber is shown in Figure 8.11. The code fragment in Listing 8.27 details the changes required for communicating on multiple sockets. Code specific to an operating system is encapsulated within `#ifdef ... #endif` code blocks.

Figure 8.11 News Subscriber with Threaded Message Processing



Listing 8.27 News Subscriber with Threaded Message Processing

```
#include <XPCMsg.h>       // Defines the XPCMessage and XPCBodyComponent
#include <Priority.h>     // Assigns definitions to news priority values
#include <iostream.h>

// The appropriate thread class is defined
#include <XPCThread.h>

void *vNewsThread(void *arg)
```

```
{
   try
   {
      // The XPCMessage object passed to the thread is recasted
      XPCMessage *newMessage = (XPCMessage *)arg;

      // See Listing 8.24 for processing, formatting and displaying a
      // XPCMessage news message.
   }
   catch(XPCException &exceptOb)
   {
       // Errors processing the news message are caught and displayed to the
       // user
       cerr << "Error processing news message: " << exceptOb.sGetException()
            << endl;
       return 0;
   }

   return 1;
}

void vProcessNews(XPCTcpSocket *_Socket, int _iNumComponents, char *_sMsgName)
{
   try
   {
      // A XPCMessage object is constructed using message components sent
      // over the network
      XPCMessage newMessage(_iNumComponents, _Socket, _sMsgName);

      // A thread is spawned and passed the XPCMessage object. The thread
      // processes and displays the received news message. Once the thread is
      // spawned vProcessNews() returns
#ifdef UNIX
      XPCPthread<int> msgThread(vNewsThread, (void *)&newMessage);
#else
      XPCThread msgThread(vNewsThread, (void *)&newMessage);
#endif
   }
   catch(XPCException &exceptOb)
```

```
    {
        // All message processing errors are caught and displayed to
        // the user
        cerr << "Error receiving news message from "
             << _sMsgName << ": " << exceptOb.sGetException() << endl;
    }
}

main(int argc, char *argv[])
{
    // This function connects to PubSubServer, subscribes to messages, and
    // receives published XPCHeader objects. See main() in Listing 8.24 for a
    // full listing
}
```

Asynchronous Message Retrieval Within a Windows Application

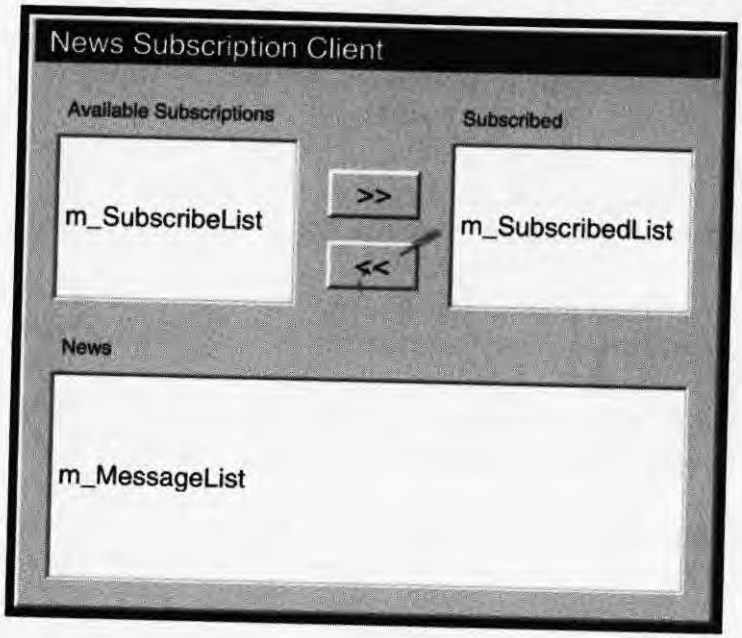
Receiving published messages within a Windows application must be performed differently because published messages suspend all Windows event processing if message retrieval is within the same thread as the main Windows program. An obvious solution is to subscribe to messages and thread `XPCMessage::vGetMessages()`. Threading `vGetMessages()` enables Windows event processing to take place while waiting to receive published messages.

Although threading `vGetMessages()` is a viable solution, it is not the ideal solution. Threading adds additional complexity because semaphores must protect global resources. A better solution is to use Windows asynchronous sockets, as discussed in Chapter 5 of *Making UNIX and Windows NT Talk*, and maintain socket and Windows processing within the main thread. Containing all event processing within the main thread avoids the complexity associated with semaphores.

Windows asynchronous sockets add socket processing to the Windows event loop. Socket communication is queued and handled in the same queue as all other Windows events. This means applications are still controlled by the user while Windows processes socket messages within the main thread.

The Windows version of the News Subscriber (Figure 8.12) allows the user to subscribe and unsubscribe to news services at any time. This is unlike the previous subscribers, which only allowed messages to be subscribed to at the beginning of the process. In order to change subscriptions, the previous applications had to be killed and restarted. Messages received within the Windows client are displayed within a scrolling list box.

Figure 8.12 The Graphical User Interface News Subscriber



The main News Subscriber dialog class (`CNewsSubClientDlg`) contains two `XPCTcpSocket` objects. Each object is dedicated to subscribing to a specific message. When messages are published, they are received on the same `XPCTcpSocket` object on which they were subscribed. Along with two `XPCTcpSocket` objects, two `XPCAsyncTcpSocket` objects are also defined. After an `XPCTcpSocket` object is constructed, connected to the `PubSubServer`, and subscribed, it is used to construct an `XPCAsyncTcpSocket` object. The `XPCAsyncTcpSocket` object enables socket processing to queue up with Windows event processing. Also included within `CNewsSubClientDlg` are methods to subscribe and unsubscribe to messages, as well as public data members associated with the Windows controls. The `CNewsSubClientDlg` class definition is shown in Listing 8.28.

Listing 8.28 The `CNewsSubClientDlg` Class Definition

```
#include <XPCAsyncTcpSocket.h>   // Defines the XPCAsyncTcpSocket class
#include <XPCMsg.h>              // Defines XPCMessage and XPCBodyComponent
classes
#include <Priority.h>     // Defines the priority values of news messages

class CNewsSubClientDlg : public CDialog
{
   XPCTcpSocket *BusSocket;    // XPCTcpSocket objet used to communicate
                               // BUSINESS messages
```

```
    XPCTcpSocket *TechSocket;   // XPCTcpSocket object used to communicate TECH
                                // messages

    // Asynchronous socket used to receive BUSINESS messages
    XPCAsyncTcpSocket<CNewsSubClientDlg> *BusAsyncSocket;

    // Asynchronous socket used to receive TECH messages
    XPCAsyncTcpSocket<CNewsSubClientDlg> *TechAsyncSocket;
public:
    // Processes asynchronous messages received from the socket
    void vProcessMessage(char *, XPCAsyncTcpSocket<CNewsSubClientDlg> *);

    // Processes asynchronous socket error messages
    void vProcessError(char *_sErrMsg)
    {
       MessageBox(_sErrMsg, "Socket Error", MB_OK);
    }

    // Methods required by XPCTcpAsyncSocket are defined but not used
    void vProcessConnection(struct sockaddr_in *addr) { return; }
    char *sGetMessageBuffer(int _iNumBytes) { return NULL; }
    void vProcessClose(struct sockaddr_in *addr) { return; }

    CNewsSubClientDlg(CWnd* pParent = NULL);   // standard constructor

    // Dialog Data
    //{{AFX_DATA(CNewsSubClientDlg)
    enum { IDD = IDD_NEWSSUBCLIENT_DIALOG };
    CListBox   m_SubscribedList;   // List box containing the list of messages
                                   // subscribed
    CListBox   m_SubscribeList;    // List box containing the list of messages
                                   // not subscribed
    CListBox   m_MessageList;      // List box containing the formatted news
                                   // messages
    …
protected:
    HICON m_hIcon;
```

```
    // Generated message map functions
    //{{AFX_MSG(CNewsSubClientDlg)
    virtual BOOL OnInitDialog();      // Method called when the News Subscription
                                      // dialog initializes
    …
    afx_msg void OnAddButton();       // Method called for subscribing to a chosen
                                      // message
    afx_msg void OnRemoveButton();    // Method called for subscribing to a chosen
                                      // message
    afx_msg void OnExitButton();      // Method called to exit the News
                                      // Subscription dialog application
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

When the Windows version of the News Subscriber initializes, the `OnInitDialog()` method of `CNewsSubClientDlg` (Listing 8.29) is called. It adds the types of message subscriptions to the `m_SubscribeList` list box control and initializes the WinSock library.

Listing 8.29 The `OnInitDialog()` Method of `CNewsSubClientDlg`

```
BOOL CNewsSubClientDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    …

    // All message types are added to the list of unsubscribed messages
    m_SubscribeList.AddString("BUSINESS");
    m_SubscribeList.AddString("TECH");

    // The WinSock library is initalized
    if (!AfxSocketInit())
    {
        MessageBox("Error initializing WinSock. Exiting Application",
                   "WinSock Error", MB_OK);
        CDialog::OnOK();
    }

    return TRUE;  // return TRUE unless you set the focus to a control
}
```

Messages are subscribed to by clicking the >> button and unsubscribed by clicking the << button. When subscribing to a message, the user chooses a message name from the m_SubscribeList and clicks the >> button to add it to m_SubscribedList. This creates an XPCTcpSocket instance that is associated with the chosen message name. The XPCTcpSocket object is created using socket port 6800 and is connected to the host running PubSubServer. Once connected, the XPCTcpSocket object is used to construct an XPCAsyncSocket object so that incoming messages are processed along with Windows events. An XPCMessage object is then created and passed a pointer to the XPCTcpSocket object, along with the name of the message being subscribed. The vSubscribe() method of the XPCMessage instance is called in order to subscribe to PubSubServer. If the subscription is successful, the message name is removed from m_SubscribeList and added to m_SubscribedList. The code for subscribing to a message is shown in Listing 8.30.

Listing 8.30 Subscribing to a Chosen Message

```
void CNewsSubClientDlg::OnAddButton()
{
   CString sSubscription;        // Stores the name of the subscribed message
                                 // chosen from m_SubscribeList

   try
   {
      // The user-chosen message is extracted from the m_SubscribeList
      int iCurSel = m_SubscribeList.GetCurSel();
      if (iCurSel == LB_ERR)
      {
         MessageBox("A Subscription Must Be Chosen", "Subscribe Error", MB_OK);
         return;
      }

      m_SubscribeList.GetText(iCurSel, sSubscription);

      if (sSubscription == "BUSINESS")
      {
         // If the user subscribed to BUSINESS, BusSocket is constructed
         // using socket port #6800 and connected to the PubSubServer
         BusSocket = new XPCTcpSocket((long int)6800);
         BusSocket->vConnect("PubSubServer_Host");

         // The BusAsyncSocket object is constructed using BusSocket as its
         // underlying means of communication.  BusAsyncSocket is defined to
         // retrieve XPCHeader objects with a specification of MSG_WAITALL.
```

```
        // MSG_WAITALL prevents the vProcessMessage() method from being
        // called until the entire XPCHeader object is received.
        BusAsyncSocket = new XPCAsyncTcpSocket<CNewsSubClientDlg>(
                    (CNewsSubClientDlg *)this, BusSocket->iGetSocketFd(),
                    sizeof(XPCPubSubMsg), MSG_WAITALL);

        // BusSocket is attached to the Windows queue
        BusAsyncSocket->vAttachSocket();

        // A XPCMessage object is constructed using the name of the
        // user-chosen message and a type of SUBSCRIBE.
        XPCMessage Subscribe(0, (char *)(const char *)sSubscription,
                          SUBSCRIBE, BusSocket);

        // The user-chosen message is subscribed
        Subscribe.vSubscribe();
    }
    else
    {
        // If the user subscribed to TECH, TechSocket is constructed
        // using socket port #6800 and connected to the PubSubServer
        TechSocket = new XPCTcpSocket((long int)6800);
        TechSocket->vConnect("PubSubServer_Host");

        // The TechAsyncSocket object is constructed using TechSocket as
        // its underlying means of communication.  TechAsyncSocket is
        // defined to retrieve XPCHeader objects with a specification of
        // MSG_WAITALL. MSG_WAITALL prevents the vProcessMessage() method
        // from being called until the entire XPCHeader object is received.
        TechAsyncSocket = new XPCAsyncTcpSocket<CNewsSubClientDlg>(
                    (CNewsSubClientDlg *)this, TechSocket->iGetSocketFd(),
                    sizeof(XPCHeader), MSG_WAITALL);
        TechAsyncSocket->vAttachSocket();

        // A XPCMessage object is constructed using the name of the
        // user-chosen message and a type of SUBSCRIBE.
        XPCMessage Subscribe(0, (char *)(const char *)sSubscription,
                          SUBSCRIBE, TechSocket);
```

```
        // The user-chosen message is subscribed
        Subscribe.vSubscribe();
    }

    // If the message is successfully subscribed, it is removed from
    // m_SubscribeList and added to m_SubscribedList
    m_SubscribeList.DeleteString(iCurSel);
    m_SubscribedList.AddString(sSubscription);
  }
  catch(XPCException &exceptObject)
  {
    // All communication errors are caught and displayed to the user
    MessageBox(exceptObject.sGetException(), "Communication Error", MB_OK);
    return;
  }
}
```

The user can also unsubscribe from a message. To unsubscribe, the user selects the message to unsubscribe from within the `m_SubscribedList` list box control and clicks the << button, which causes the `OnRemoveButton()` method to be called. This method deletes the `XPCAsyncTcpSocket` object associated with the chosen message then deletes the associated `XPCTcpSocket` object. The objects must be deleted in this order because `XPCAsyncTcpSocket` inherits `XPCTcpSocket`. Deleting the objects disconnects the socket connection associated with a message. When the socket disconnects, `PubSubServer` detects the loss of connection and removes the client from its list of subscriptions. Because separate socket connections are used for each message, unsubscribing from one message does not eliminate subscriptions from other messages. `PubSubServer` only removes the subscription associated with the disconnected socket. If the socket successfully disconnects, the message name is removed from the `m_SubscribedList` list box control and added to the `m_SubscriptionList` list box control. The code for unsubscribing from a chosen message is shown in Listing 8.31.

Listing 8.31 Unsubscribing From a Chosen Message

```
void CNewsSubClientDlg::OnRemoveButton()
{
  CString sSubscription;      // Stores the name of the message that is being
                              // unsubscribed

  try
  {
    // The user-chosen message is extracted from m_SubscribedList
    int iCurSel = m_SubscribedList.GetCurSel();
```

```
        if (iCurSel == LB_ERR)
        {
            MessageBox("A Subscription Must Be Chosen", "UnSubscribe Error",
                       MB_OK);
            return;
        }
        m_SubscribedList.GetText(iCurSel, sSubscription);

        // If subscription to unsubscribe is BUSINESS, the BusAsyncSocket and
        // BusSocket objects are deleted; otherwise, the TechAsyncSocket and
        // TechSocket objects are deleted. Deletion causes the specific socket
        // connection with PubSubServer to terminate and therefore the News
        // Subscription client receives no more messages.
        if (sSubscription == "BUSINESS")
        {
            delete BusAsyncSocket;
            delete BusSocket;
        }
        else
        {
            delete TechAsyncSocket;
            delete TechSocket;
        }

        // Message unsubscribed from is removed from m_SubscribeList and added
        // to m_SubscribedList
        m_SubscribeList.AddString(sSubscription);
        m_SubscribedList.DeleteString(iCurSel);
    }
    catch(XPCException &exceptObject)
    {
        // All communication errors are caught and displayed to the user
        MessageBox(exceptObject.sGetException(), "Communication Error", MB_OK);
        return;
    }
}
```

Messages can be subscribed to and unsubscribed from at any time. This feature is made available by asynchronous sockets, which enable Windows events and socket messages to be queued together and contained within the same thread of execution. The user can manipulate the Windows interface while socket communication takes place. Unlike previous subscription

client examples, there is no need to contain the retrieval of published methods within a loop. Asynchronous sockets place the retrieval of socket messages within its event loop. Retrieving socket messages in an event loop does not manipulate the thread in which it is executing; therefore, the user still has control over the Windows interface.

When published messages are retrieved, the `OnReceive()` method of `XPCAsyncTcpSocket` is called. This method receives the first part of a published message that is the `XPCHeader` object. `OnReceive()` places the `XPCHeader` object within a character buffer allocated to the size specified within `OnAddButton()`. The `MSG_WAITALL` TCP socket option was specified when constructing the `XPCAsyncTcpSocket` object so that it will receive the entire `XPCHeader` object before continuing. It is important to specify the retrieval of the entire message because TCP sockets are sent as streams. `OnReceive()` returns if it has any portion of the message.

`OnReceive()` calls the `vProcessMessage()` method of `CNewsSubClientDlg` (Listing 8.32). Because `CNewsSubClientDlg` defines `CAsyncTcpSocket` instances, `CAsyncTcpSocket` must contain a `vProcessMessage()` method, which receives the character buffer containing the `XPCHeader` object. `vProcessMessage()` casts the character buffer to an `XPCHeader` object and calls `vProcessNews()` to process the remainder of the incoming message. The message subject of `XPCHeader` is extracted and `vProcessNews()` is passed the appropriate `XPCTcpSocket` object to receive the specified message. Also passed to `vProcessNews()` is the number of `XPCMessage` objects associated with the message and the name of the message being received. `vProcessMessage()` is the asynchronous socket replacement for the `XPCMessage vGetMessages()`.

Listing 8.32 The `vProcessMessage()` Method

```
void CNewsSubClientDlg::vProcessMessage(char *sMsg, int iNumBytes)
{
   // Character buffer sent from XPCAsyncTcpSocket is cast to an
   // XPCHeader object
   XPCHeader *Header = (XPCHeader *)sMsg;

   if (strcmp(Header->sGetSubscription(), "BUSINESS") == 0)
      // If the message name contained within Header is BUSINESS, the
      // vProcessNews() function is called to process the remainder of the
      // message. vProcessNews() is passed a pointer to BusSocket.
      vProcessNews(BusSocket, Header->iNumComponents(),
                   Header->sGetSubscription());
   else
      // If the message name contained within Header is TECH, the
      // vProcessNews() function is called to process the remainder of the
      // message. vProcessNews() is passed a pointer to TechSocket.
      vProcessNews(TechSocket, Header->iNumComponents(),
                   Header->sGetSubscription());
}
```

In previous examples, vProcessNews() was the user-defined function passed to vGetMessages(). The vProcessNews() method in Listing 8.33 is similar to its previous versions, except that the news message received is placed within the m_MessageList list box control of CNewsSubClientDlg instead of displayed on the console window. vProcessNews() constructs an XPCMessage object using the XPCTcpSocket object, the number of incoming XPCBodyComponent objects, and the name of the incoming message. The XPCMessage object returns from construction when all XPCBodyComponent objects are received or an error occurs. When the entire XPCMessage object is constructed, its values are extracted, formatted within a character string, and inserted within the m_MessageList list box.

Listing 8.33 The vProcessNews() Function

```
void vProcessNews(XPCTcpSocket *_Socket, int _iNumComponents, char *_sMsgName)
{
   // A pointer to the main dialog window is retrieved
   CNewsSubClientDlg *dlgPtr = (CNewsSubClientDlg *)AfxGetMainWnd();
   try
   {
      // The remainder of the message is retrieved
      XPCMessage newMessage(_iNumComponents, _Socket, _sMsgName);

      char sNewsItem[1024];   // Holds the formatted news message
      int iPriority;          // Holds the priority value of the news message
      char sPriority[256];    // Holds the news priority string representation
      char sNews[256];        // Holds the news message
      char sSource[256];      // Holds the news source

      // The priority of the news item is extracted.
      if (!newMessage.iGetValueByName("Priority", &iPriority))
         iPriority = UNKNOWN;

      // The news message is extracted
      if (!newMessage.iGetValueByName("News", sNews))
         strcpy(sNews, "UNKNOWN");

      // The news source is extracted
      if (!newMessage.iGetValueByName("Source", sSource))
         strcpy(sSource, "UNKNOWN");

      // A string representation of the priority is stored based on the
      // priority value extracted from the XPCMessage object
```

```
        switch(iPriority)
        {
           case UNKNOWN:
              strcpy(sPriority, "UNKNOWN");
              break;
           case CRITICAL:
              strcpy(sPriority, "Critical");
              break;
           case IMPORTANT:
              strcpy(sPriority, "Important");
              break;
           case INFORMATIVE:
              strcpy(sPriority, "Informative");
              break;
           }

        // The entire news message is formatted and displayed to the user.
        sprintf(sNewsItem, "%s : From %s Issued by %s - %s",
                sPriority, _sMsgName, sSource, sNews);

        // Formatted news message is inserted into the m_MessageList list box
        dlgPtr->m_MessageList.InsertString(0, sNewsItem);
     }
     catch(XPCException &exceptOb)
     {
        // All message processing errors are caught and displayed to the user
        char sMsg[512];

        sprintf(sMsg, "Error receiving news message from %s: %s",
                _sMsgName, exceptOb.sGetException());
        dlgPtr->MessageBox(sMsg, "News Error", MB_OK);
     }
  }
```

8.6 Conclusion

The publish-and-subscribe architecture has the advantages of allowing for dynamic connections, with multiple subscribers and publishers communicating using the same message, and having the reliability of TCP socket communication. The process that facilitates these features is the Subscribe server. The Subscribe server is responsible for keeping track of all processes subscribing to a particular message and publishing the message to those processes when one is received. The Subscribe server is the entry point for all processes within the publish-and-subscribe system and it must be able to handle both UNIX and Windows NT data since processes can connect from either operating system.

Generic messages allow for run-time message creation. The structure of the message being sent and received is dynamic and can be changed based on run-time parameters. In previous systems this flexibility was not available. All processes were required to know all data structures being communicated. By creating message structure at run time, the communicating processes don't need to be updated and recompiled each time the message structure has changed.

The reliability of the publish-and-subscribe architecture and the flexibility of the generic message framework make this system an attractive approach for cross-platform communication development.