Object-Oriented Distributed and Parallel I/O Streams

Andrew J. Dick

A thesis submitted to the Faculty of Graduate Studies
in partial fulfillment of the
requirements for the degree of

Master of Science

Department of Computer Science
York University, North York, Ontario

26 February 1999

Abstract

Writing programs for parallel and distributed computing environments can be significantly more complex than writing programs for their sequential counterparts. These complexities mainly arise from the additional synchronization and communication requirements imposed by such environments. These requirements also make debugging and maintaining such programs significantly more complicated. The problems of debugging and maintenance are further exacerbated by the lack of good debuggers and the lack of proper I/O support for such environments.

The goal of this thesis is to design and implement an object-oriented C++ streams library (`piostream`) which provides convenient and extensible constructs for input and output in parallel and distributed programming environments. These environments include multi-threaded applications, multiprocessors, and distributed systems. The `piostream` library is based on the C++ `iostream` library, thus simplifying the use of I/O operations in parallel and distributed environments. A prototype implementation is described and used to demonstrate the feasibility of the `piostream` library design and the ease with which it can be used.

Acknowledgements

I dedicate this thesis to the memory of my mother, who taught me what perseverance really is, through her battle with diabetes. I thank my family for all their loving support. Much appreciation goes to both my supervisors, Eshrat and Tim, for their wisdom and guidance throughout my graduate years. A special thanks to Bill O'Farrell and Greg Wilson, who both provided the initial framework for my library, and also patiently provided me with technical information. Most of all, I thank Samantha, without whose love and support, this thesis would not have been possible.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

To perform useful calculations, programs often require data that must be obtained from an input device or file. Subsequently, the program results must be written to an output device or file. In a networked environment, programs that perform I/O must buffer and transmit data to or from tasks executing on remote machines to tasks executing on machines that have access to the desired I/O devices and files. In distributed and parallel programs, the standard I/O devices and files are accessible from the machine where the user program initially begins execution. This machine is called the main host, and the task that executes initially on the main host is called the main program. The standard I/O devices consist of two output devices: standard output (`stdout`) and standard error (`stderr`) and one input device: standard input (`stdin`). Sending and receiving data between user processes that may be executing on remote machines and the main host I/O devices requires explicit buffering, transferring and coordination of data. Access to the main host I/O devices must be synchronized by the user to support the desired behaviour. The added complexity to user programs lowers portability and maintainability which makes writing and debugging programs more difficult.

Significant research efforts have been expended in recent years to improve the performance of I/O subsystems by using parallel techniques to transfer portions of data to and from several storage devices simultaneously. These efforts have concentrated almost exclusively on alleviating the I/O performance bottleneck by using multiple disk devices to perform file I/O[10, 15]. Unfortunately, techniques for providing users with the ability to simply and easily perform input and output operations on multiple processors or hosts simultaneously for the purposes of debugging, executing and maintaining parallel and distributed programs have received little attention.

This thesis addresses this shortcoming and focuses on techniques that enable users to write parallel and distributed applications that perform I/O. The `piostream` library is not intended to improve the performance of parallel and distributed programs. Instead it is designed to provide simple and easy to use I/O constructs for the purpose of writing, debugging, executing and maintaining parallel and distributed programs. By using the `piostream` library, programs can execute input and output operations from any machine in a networked environment and the input and output data is transparently obtained from or sent to the originating host by the underlying run-time system. The target environments of the library include multiprocessors, networks of workstations, and distributed workstations and PCs.

1.2 Objectives

This thesis describes our design and implementation of a parallel streams library called `piostream`. We discuss the issues involved in designing and implementing the components of the `piostream` library. The `piostream` library is designed specifically to permit parallel and distributed programs to use common input and output operations regardless of the machine on which the operations are being performed. The `piostream` library is comprised of the following components: `postream` for parallel output to `stdout` and `stderr`, `pistream` for parallel input from `stdin`, and `pfstream` for parallel file I/O. The `pfstream` component, similar to the C++ `fstream` library, consists of three different classes that can be instantiated for file I/O. The `fstream` library has an input file stream (`ifstream`), an output file stream (`ofstream`), and a bi-directional file I/O stream (`fstream`). Similarly the `pfstream` library supports: `pifstream` for input, `pofstream` for output, and `pfstream` for bi-directional file I/O. A prototype has been implemented for the `postream` and `pistream` components of the `piostream` library. The `pfstream` component has been designed but not implemented.

In this thesis, much of the discussion of the design of the `piostream` library takes place in the context of a parallel system that provides concurrency through an active object model such as ABC++ [2, 16]. This design and the `piostream` library could easily be implemented using any object-oriented concurrency model. Because the `piostream` library constructs are meant for use in object-oriented parallel and distributed programs, they must also satisfy a number of object-oriented design constraints. The issues of proper interface design, encapsulation, extensibility and efficiency must be considered throughout the design and implementation of the `piostream` library.

Encapsulation deals with information hiding and data abstraction. A large portion of the cost of software development is attributed to software maintenance due to complex and intertwined programs [14, 23]. Encapsulation attempts to rectify this problem by grouping an object's data and methods together. Encapsulation enforces well-defined interfaces for accessing and modifying an object's data members. This encapsulation permits the user to use the class without knowledge of its implementation, and subsequently allows the implementation to be changed without affecting the user [3][1]. By simple application of this concept, the `piostream` library's implementation details should be hidden from the user of the library. The user should not require significant modifications to their code, beyond the normal changes required to convert to the parallel paradigm. In addition to requiring minimal interface changes, any additional objects and methods required to implement the `piostream` library should be completely abstracted from the user. The user should be solely concerned with how to create and utilize the methods of user objects and not with the `piostream` library run-time operations. Consequently it is of primary importance that the `piostream` interface be as close to the existing standard C++ `iostream` interface as possible.

One of the main advantages of the C++ `iostream` library is its extensibility. In addition to being able to output predefined data types, the user can output user-defined data types by overloading `operator<<()`. Figure 1.1A defines a simple class, `complex`, that has both a `print()` method and an extensible `operator<<()` function. Figure 1.1B illustrates how `operator<<()` functions can be overloaded to support output of the `complex` class with the `piostream` library. Figure 1.1C shows the different syntax required to output a `complex` object using the `iostream` library and the `piostream` library. Both the extensible (`operator<<()`) and non-extensible (`print()`) approaches to output are illustrated for the `iostream` library. Clearly the extensible interface simplifies the user code and hence, should be supported in the `piostream` library. Subsequently the `piostream` library supports input and output of arbitrary objects through the overloading of `operator<<()` and `operator>>()`. The `piostream` library supports user-defined manipulator functions in addition to all existing `iostream` manipulator functions.

The issue of efficiency becomes critical in a parallel system due to the potential amount of synchronization required for communication between parallel and distributed objects. Several potential bottlenecks exist during the transfer of data from multiple objects to a single source destination, or conversely, from a single source to multiple objects. A concern during the design of the `piostream` library is minimizing the impact of these bottlenecks on performance. The `piostream` library does this by limiting the use of shared locks and synchronous messaging in the design of the `piostream` library components. Both shared locks and synchronous messaging increase the length of time tasks block during communication and synchronization. This blocking reduces the parallelism of the program and hence adversely impacts

[1] Assuming the external interface remains the same.

(A)

```
class complex {
public:
  // Standard class methods
 •••
  // Print function
  void print()
  {
    cout << real << " + "
         << imag << "i";
  }
private:
  float real;
  float imag;
  // Friends
  friend ostream &operator <<(ostream &os,
                  const complex & data);
};
// Friend ostream operator - outputs in form "real + imag i"
// For example: 5.2 + 6.3i
ostream &operator <<(ostream &os,
                  const complex & data)
{
  os << real << " + " << imag << "i";
  return os;
}
```

(B)

```
class complex {
public:
  // Standard class methods
 •••
private:
  float real;
  float imag;
  // Friends
  friend postream &operator <<(postream &pos,
                  const complex & data);
  friend ostream &operator <<(ostream &os,
                  const complex & data);
};
// Friend postream operator - outputs in form "real + imag i"
// For example: 5.2 + 6.3i
postream &operator <<(postream &pos,
                  const complex & data)
{
  pos << real << " + " << imag << "i";
  return pos;
}
// Friend ostream operator - outputs in form "real + imag i"
// For example: 5.2 + 6.3i
ostream &operator <<(ostream &os,
                  const complex & data)
{
  os << real << " + " << imag << "i";
  return os;
}
```

(C)

```
// Create complex object
Complex c(5.2, 6.3);

// Output using non-extensible print function
cout << "The complex number is ";
c.print();
cout << endl;

// Output using extensible overloaded operator<<
cout << "The complex number is " << c << endl;

// Output using extensible parallel output construct
pout << "The complex number is " << c << endl;
```

Figure 1.1: This diagram illustrates the benefits of extensible output. (A) defines a class `complex` that has a built-in `print()` method, and an extensible `operator<<()` friend function. (B) depicts the same class `complex` with the `piostream` library's overloaded `operator<<()` function. (C) illustrates the different syntax required to output the class' data using each approach.

the program's performance.

1.3 Contributions

The primary advantages of the `piostream` library and its constructs and thus the main contributions of this thesis are:

- They are based on the C++ `iostream` library [21] and as a result are familiar and straightforward to use as well as easy to extend. Since high-level constructs are provided, users are not required to construct, send and receive elaborate messages. The programmer simply uses `pin` and `pout` constructs in the same way that sequential C++ programs currently use `cin` and `cout` for I/O. The user can easily input and output predefined and user-defined data types, by overloading `operator<<()` or `operator>>()`.

- They are able to support C++ I/O manipulator functions which allow precisely formatted output. Extensible user-defined manipulator functions are also supported.

- They are built using and are fully compatible with standard C++, which means that no special compiler support or language extensions are required.

- The library provides mechanisms for identifying the source of the output. For example, output being printed by tasks executing on several remote client machines is automatically and transparently collected, collated, and printed to a screen or file on the originating host along with information identifying the host name, process id, and thread id of the task performing the output.

- Input constructs can be used on hosts that are potentially remote. The `pistream` interface permits programs to simply use the `pin` object to obtain input when desired. The `piostream` library transparently provides the facilities to read input on remote hosts with a similar interface as the C++ `iostream` library. Input can be distributed to each requesting object by using a single shared stream position or multiple independent stream positions.

- Synchronization is an integral part of the library and as a result the user is not required to synchronize access to input and output streams. The proposed file I/O constructs transparently provide mutually exclusive access to files.

- The proposed file I/O constructs allow the user to access the main host's file system from remote hosts for input and output.

One of the main contributions of the `piostream` library is its clean and simple interface. Many parallel programs are developed from original sequential versions. As a result, converting a program from a sequential program to a parallel program can be complicated and time consuming. The ideal constructs for parallel programs should provide the same interface as their sequential counterparts where possible. The `piostream` library interface is modeled on the existing standard C++ `iostream` library [21]. The `piostream` library could be modified to conform to the new standards once they are released. To illustrate the ease with which `piostream` can be used for output, Figure 1.2 provides an example in C++ using the C++ `iostream` interface for `cout`, `cerr`, `cin` and `fstream` and their `piostream` counterparts `pout`,

(A)

```
// Counter
int num_elements;
// Data
int element;
// Input file
ifstream in("input", ios::exist);
// Output file
ofstream out("output", ios::nocreate);
if ( (!in) || (!out) ) {
  cerr << "Error in input or "
       << "output file." << endl;
  exit(0);
}
// Header buffer
char header[256];

// Get number of input elements
cout << "Please input number of "
     << "elements:" << flush;
cin >> num_elements;

// Get header for output file
cout << "Please enter header "
cout << "for output file." << endl;
cin.getline(header, 256);

// Insert header into output file
out << header;

// Copy and convert decimal data from input file
// to hexadecimal in output file
for (int i = 0; i < num_elements; i++) {
  in >> ws >> element;
  out << hex << element;
}
```

(B)

```
// Counter
int num_elements;
// Data
int element;
// Input file
pifstream in("input", ios::exist);
// Output file
pofstream out("output", ios::nocreate);
if ( (!in) || (!out) ) {
  perr << "Error in input or "
       << "output file." << endl;
  exit(0);
}
// Header buffer
char header[256];

// Get number of input elements
pout << "Please input number of "
     << "elements:" << flush;
pin >> num_elements;

// Get header for output file
pout << "Please enter header "
pout << "for output file." << endl;
pin.getline(header, 256);

// Insert header into output file
out << header;

// Copy and convert decimal data from input file
// to hexadecimal in output file
for (int i = 0; i < num_elements; i++) {
  in >> ws >> element;
  out << hex << element;
}
```

Figure 1.2: This diagram provides a comparison of interfaces between the C++ `iostream` library and the `piostream` library. (A) illustrates the standard C++ `iostream` interface for using standard input (`stdin`), standard output (`stdout`), standard error (`stderr`) and file I/O. (B) illustrates the `piostream` library interface for similar code. `Stdin`, `stdout`, `stderr`, and the file system are all located on the main host in each example. Both examples first instantiate several objects including input and output file streams. The input and output file stream's state is checked, and if an error is detected, an error message is written to `stderr`. Output is written to `stdout` and input is read from `stdin`. Finally, data is transferred from the input file to the output file, including a user-input header comment. The use of I/O chaining, multiple data types and manipulator functions is also demonstrated.

`perr`, `pin`, and `pfstream`. The C++ `iostream` interface is illustrated in Figure 1.2A and the `piostream` interface is shown in Figure 1.2B.

In this trivial example, a remote process is used to read decimal numbers from a source file and write each number's corresponding hexadecimal value to a second destination file. Both the source and destination files are located on the main host's file system. The user provides both the number of elements to be converted from decimal to hexadecimal, and a header comment for the output file. If either the input or output files are invalid, a message is written to `stderr` on the main host. The header is first written to the output file. The required number of decimal integers are read from the input file, ignoring the white space between each integer using the manipulator function `ws`. Each decimal number is converted to hexadecimal using the manipulator function `hex`, and written to the output file. The modifications required in order to move from the standard C++ `iostream` interface to the parallel specific I/O interface are to use the `pout`, `perr`, `pin`, and `pfstream` objects rather than their sequential counterparts, `cout`, `cerr`, `cin`, and `fstream`.

1.4 Thesis Outline

Chapter 2 of the thesis describes some popular parallel and distributed systems and examines the support they provide for parallel and distributed I/O. It also examines some existing parallel I/O systems and discusses their strengths and weaknesses. Relevant background is provided on the active object concurrency model, and the implementation environment, ABC++. A brief discussion of the C++ `iostream` library is provided to illustrate the desired interface and behaviour of the `piostream` library.

Chapters 3 and 4 provide a discussion of the `postream` and `pistream` components of the `piostream` library. These chapters describe the problems and issues involved in designing the `postream` and `pistream` components. Chapter 3 discusses the strengths and weaknesses of various designs considered for the `postream` component, before providing an overview of the model chosen, and a description of the implementation architecture. Chapter 4 discusses the design and implementation issues involved in supporting the same model for the `pistream` component.

Chapter 5 discusses the issues involved in designing the `pfstream` construct for remote file I/O. A high-level design is provided for the `pfstream` construct, including possible solutions to the major issues. The `pfstream` component is not implemented. The thesis conclusions and possibilities for future work are presented in Chapter 6.

Chapter 2

Background

Distributed I/O solutions that address the problems of debugging and maintenance of parallel and distributed programs have received little attention in the literature and even less research has been conducted on object-oriented solutions to this problem. This chapter first describes the C++ `iostream` architecture and interface which is considered high-level, extensible, and easy to use. The `iostream` interface is presented as a model for developing the `piostream` library interface. The `iostream` library provides several manipulator functions that support precise formatting of input and output data. A brief overview of the `iostream` manipulator functions is provided to illustrate the problems inherent in extending the functions for use with parallel and distributed programs. The chapter then examines how distributed I/O is supported by the popular PVM (Parallel Virtual Machine) system [7] and the MPI (Message Passing Interface) standard [18]. The section also describes distributed file I/O using the NFS (Network File System) [20] and examines the Condor High Throughput Computing System's [13] use of RU (Remote Unix) [12] to support distributed I/O. Both the NFS and RU systems are potential platforms that could be encapsulated by the `piostream` library to support distributed I/O. The chapter also describes an existing object-oriented solution for parallel I/O, called `pC++/streams` [9]. The active object model of concurrency and the implementation environment, ABC++, are described to provide an understanding of the framework in which the `piostream` library is implemented.

2.1 Iostream Library

The `iostream` library simplifies user I/O operations so that programs are easier to write, debug, and maintain. The `iostream` library is high-level and object-oriented, providing an extensible and easy to use I/O interface that promotes portability and maintainability. This section describes the C++ `iostream` library components including the output stream (`ostream`), the input stream (`istream`), the file stream classes (`ofstream`, `ifstream`, and `fstream`), and the use of the `iostream` manipulator functions. The `iostream` library provides several high-level constructs for I/O purposes: `cout` for output to `stdout`; `cerr` for output to `stderr`; and `cin` for input from `stdin`. The `fstream` library provides several classes that can be instantiated for file I/O purposes.

Each `iostream` class consists of three layers. The `streambuf` class forms the lowest layer. The `streambuf` class is a buffer that acts as an intermediary between the input and output devices and the general I/O system. The `ios` class is the middle layer which contains a `streambuf` object. The `ios` class acts as a specification system responsible for formatting and error reporting. The outer layer is a translator system that converts C++ language typed objects to and from sequences of characters using the `ostream` and `istream` classes [21]. The `fstream` class architecture is more complex and is described in more detail in Section 2.1.1

The `streambuf` class is conceptually a buffer of characters that is divided into two areas (that possibly overlap). Each area is accessed by a pointer: the *get* area (for input) is accessed by the `get` pointer and the *put* area (for output) is accessed by the `put` pointer. The size of the `streambuf` buffer can either be dynamic or statically set at instantiation. The `streambuf` buffer presents the abstraction of unlimited storage through the use of several methods that transparently move data in (`underflow()`) and out (`overflow()`, `sync()`) of the buffer's `get` and `put` areas respectively.

The `underflow()` method is invoked if data is extracted from the `streambuf` buffer and the get pointer is at the limit of the get area (e.g., there is no data to be read in the buffer). The `underflow()` method attempts to read more data from the connected input device (`stdin` for the `cin` object) and places it in the get area. If no data is available, the method returns the `EOF` value. Conversely the `overflow()` method is invoked if data is inserted into the `streambuf` buffer and the put pointer is at the limit of the put area (e.g., the buffer is full). The `overflow()` method attempts to make room in the put area by moving data from the put area to the connected output device (`stdout` for the `cout` object and `stderr` for the `cerr` object). If the data cannot be moved, the method returns the `EOF` value. The `sync()` method is invoked to move the data from the put area to the connected output device. For instance the `iostream` manipulator functions that flush the output stream (`flush` and `endl`) invoke the underlying `streambuf sync()` method. Manipulator functions are discussed in more detail in Section 2.1.2.

The `ios` specification class provides error reporting and formatting specification. The `ios` class defines various flags that specify the format state of the `ios` class. For example, the `ios` format flags can specify the base number (decimal, hexadecimal, or octal) or the notation format (scientific or fixed). The `ios` class also defines error states that specify the current state (good, bad, fail or end of file condition) of the underlying streambuf buffer.

The istream and ostream translation classes derive the functionality of the ios base class and provide translation facilities. These facilities support the translation of C++ language typed objects to and from sequences of characters. The ostream class provides an interface for the output of all predefined C++ object types. The operator<<() friend functions encapsulate the output translation facilities of the ostream class. The istream class similarly provides an interface for the input of all predefined object types. The istream class provides several operator>>() friend functions that encapsulate the input translation facilities of the istream class. The istream class also provides several get() and getline() methods that extract one or more characters, up to the specified delimiting character from the streambuf buffer and insert them into the provided buffer argument. The default delimiting character is '\n', the new line character. The difference between the get() and getline() methods is that the get() methods do not extract the delimiting character from the streambuf buffer whereas the getline() methods extract and discard the delimiter.

2.1.1 File Streams

The C++ iostream library provides a stream interface for file I/O called the fstream library. The fstream library makes manipulating files in C++ easier and safer than in C because of its simple and type-safe interface [4] (pp. 214-218). The fstream library provides three different constructs for file I/O: ofstream for output only, ifstream for input only, and fstream for bi-directional I/O. The constructors of the fstream classes automatically open the specified file. The file can be specified by name or by a file descriptor of an already open file. The user can explicitly close the file with the fstream close() method. Alternatively, the destructor closes the file when the fstream object goes out of scope or is deleted.

The fstream constructs are composed of three layers similar to the iostream constructs. The lowest layer is the filebuf class which contains a buffer. The filebuf class acts as an intermediary between the buffer and the attached file. The fstream_common class contains a filebuf object and defines the common functionality of the input, output, and, bi-directional fstream variants. Both the istream and ostream translator functions derive the fstream_common class functionality. Each class provides translator functions for converting C++ typed objects to and from sequences of characters. The bi-directional class, fstream, derives the functionality of both the ifstream and ofstream classes.

Files can be opened with a variety of different modes, as listed in Figure 2.1. These different modes specify conditions regarding the opening and use of a file. Such conditions as read only, write only, and open an existing file only, can be set.

Open Mode	Function
`ios::in`	Opens an input file. Prevents truncation of ofstream.
`ios::app`	Opens an output file for appending.
`ios::ate`	Opens an existing file. Seeks to end of file. Input or output files.
`ios::nocreate`	Open existing file or fail.
`ios::noreplace`	Open non-existing file or fail.
`ios::trunc`	Opens a file and deletes the old file if the old file exists.
`ios::out`	Opens an output file. Without ios::app, ios::ate, or ios::in ios::trunc is implied for ofstream.

Figure 2.1: List of available open modes for files with `iostream` file I/O. Modified from [4] (p. 217).

These modes can be combined using a bitwise `OR`.

The `ofstream` object provides the same interface and methods as the `ostream` class. Similarly, the input file stream, `ifstream`, provides the same interface and methods as the `istream` class. The bi-directional `fstream` construct combines both `ifstream` and `ofstream` interfaces through multiple inheritance. The `fstream` constructs can move the file pointer within the file stream[1]. A different file pointer exists for input and output. However both file pointers are synchronized each time the file stream switches from getting to putting, providing the abstraction of a single file pointer. The input pointer position can be read with the method `tellg()`, and set with the method `seekg()`. The output pointer position can be read and set with the methods `tellp()` and `seekp()` respectively. The bi-directional class supports both the read and set position methods for each pointer.

2.1.2 Manipulator Functions

The standard C++ `iostream` library provides many formatting functions that allow precise manipulation of input and output data. Most parallel I/O facilities do not support the standard manipulator functions in a distributed environment. The following section will briefly describe the existing manipulator functions and discuss

[1] Although the `istream` and `ostream` classes have the capability to seek, the use of the `seek()` method with `cin` or `cout` is undefined [4] (p. 221).

the implementation problems involved in supporting these functions in a parallel and distributed environment.

The basic `iostream` library provides many parameterless manipulators. These manipulators perform such tasks as flushing the output stream (`endl`, `flush`), consuming white space (`ws`, `skipws`), and converting the base of a number (`dec` — decimal; `oct` — octal; `hex` — hexadecimal). Figure 2.2 lists the available manipulator functions provided by the `iostream` library. Chapter 3 shows that these manipulator functions are easily overloaded. The overloaded manipulators encapsulate the required communication facilities used to implement the `piostream` library.

Manipulator	Chained	Effect
`flush`	Y	Flushes the stream.
`endl`	Y	Adds a new line character and flushes the tream.
`hex`	Y	Changes integer to hexadecimal.
`dec`	Y	Changes integer to decimal.
`oct`	Y	Changes integer to octal.
`showbase`	N	Indicates numberic base of integer value (decimal, octal, hexadecimal).
`uppercase`	N	Display uppercase A-F for hexadecimal values and E for scientific values.
`showpos`	N	Show plus sign (+) for positive values.
`showpoint`	N	Show decimal point and trailing zeros for floating-point values.
`scientific` `fixed`	N	Use scientific notation.

`left`
`right`
`internal`
	N	Left-align, pad on right.
		Right-align, pad on left.
		Fill between leading sign or base indicator and value.

`ws` `skipws`	Y N	Consumes whitespace - same as skipws. Skip whitespace on input.

Figure 2.2: This table lists the parameterless manipulator functions available from the `iostream` library. A description of each manipulator is provided, including whether the manipulator can be chained. Manipulators that cannot be chained can be used with the `setf()` and `unsetf()` `iostream` methods. Modified from [4] (p. 243).

A second group of manipulator functions that use parameters are provided in the `iostream` manipulator library. These manipulator functions support formatting such as setting the field width (`setw`), setting the precision of numbers (`setprecision`), and the setting and resetting of the format flags (`setiosflags`, `resetiosflags`) of many of the manipulators listed in Figure 2.2. A full list of manipulator functions that use parameters are provided in Figure 2.3.

Manipulator	Effect
`setiosflags(fmtflags n)`	Sets the format flags specified by n.
`resetiosflags(fmtflags n)`	Clears the format flags specified by n.
`setbase(base n)`	Changes base to n (10, 8, 16)
`setfill(char n)`	Changes the fill character to n.
`setprecision(int n)`	Changes the precision to n.
`setw(int n)`	Changes field width to n.

Figure 2.3: This table lists the manipulator functions that use parameters available from the `iostream` manipulator library (`iomanip.h`). Note that many of the manipulators listed in Figure 2.2 can be used in a chained invocation with the manipulators `setiosflags` and `resetiosflags`. Modified from [4] (p. 244).

The parameterless manipulator functions can be used in either a direct or chained fashion by utilizing the `iostream` methods `setf()` and `unsetf()` for direct invocation, or `setiosflags` and `resetiosflags` for chained invocations. Figure 2.4 illustrates the use of a selected number of manipulators for both direct and chained invocation approaches.

Manipulator functions that use parameters are difficult to overload because of the approach used to implement them. First consider the case of a parameterless manipulator, such as `hex`. Such a manipulator is a function that takes an `ostream` parameter and returns an `ostream` object. The chained invocation is accomplished by using an `operator<<()` function that takes two parameters: an `ostream` object and a pointer to a function that itself takes an `ostream` object parameter and returns an `ostream` object. This `operator<<()` function invokes the passed function and passes the `ostream` parameter to it. The `hex` manipulator first modifies and then returns the `ostream` parameter.

The case of manipulator functions that use parameters, however is more difficult. The `operator<<()` method is a binary operator that requires exactly two parameters. Consequently, the same approach cannot be used to handle parameterized

(A) Manipulator	(B) Direct
`setprecision`	`cout.precision(val);`
`showpos`	`cout.setf(ios::showpos);`
`showbase`	`cout.unsetf(ios::showbase);`

(C) Chained

```
cout << setprecision(val)
     << setiosflags(ios::showpos)
     << resetiosflags(ios::showpoint);
```

Figure 2.4: This diagram illustrates the use of several manipulator functions in both a direct and chained invocation fashion. (B) shows the manipulator functions listed in (A) using a direct invocation. (C) shows the chained invocation of the same manipulators.

manipulators. A one parameter manipulator for example would require: an `ostream` object, the manipulator argument, and a pointer to a function that takes two parameters (the `ostream` object and the manipulator argument) and returns an `ostream` object. These three arguments cannot be transferred within a single `operator<<()` invocation.

To overcome this problem, a temporary object is used that takes the manipulator parameter as a constructor argument. The subsequent `operator<<()` invocation uses the temporary object and an `ostream` object as parameters. The temporary object defines the `operator<<()` function as a friend to allow the `operator<<()` function to access the temporary object's data member. The `ostream` object is modified using the data member, and is then returned. By this approach two pieces of information, the manipulator function name and its argument, can be contained within a single token of information. More details on this technique are provided in [4] (pp. 171-178). The problems encountered in implementing parameterized manipulator functions for use in parallel and distributed environments are described in Section 3.3.4.

2.2 Related Work

Tasks executing on remote hosts often cannot perform I/O with `stdin`, `stdout`, and `stderr` on the main host because the remote hosts are not connected to these I/O devices. In cases where the main host I/O devices are accessible by remote tasks, synchronization between the multiple tasks becomes a concern. Unsynchronized access to output devices results in the interleaving of output together. The identification of each task's output data is also a problem because the data alone often has no discernible origin identification. Both of these problems, unsynchronized data interleaving and data origin identification, contribute to difficulty in debugging parallel and distributed programs. These issues and possible solutions are described in Sections 3.1–3.2.

Alternatively, multiple objects that input from `stdin` on the main host must be coordinated to provide the desired data distribution. Possible distributions include broadcast, in which all tasks input identical data (if identical input requests are made), and striped, in which the data is distributed sequentially as each task requests data (so that no two tasks receive the same data). Striped and broadcast data distributions are discussed in more detail in Section 4.1. The above concerns provide a framework for discussion of the PVM system and MPI standard with respect to their support of remote task I/O operations.

PVM [7], which is a widely used library for parallel programming, provides no access to `stdin` on the main host for remote tasks. Each task in PVM is provided with a `stdout` sink, a construct that is inherited from its parent task. The PVM run-time system collects a child's output and transmits it to the parent task in a run-time control message. Subsequently, each parent is responsible for its children's output, with the main program eventually handling all output data which is output to either the main host `stdout`, `stderr`, or a log file (depending on the implementation). PVM identifies each task's output by transparently prepending identification information. The PVM run-time system marks the end of each task's output statement to separate the output of multiple tasks. The main drawback of the approach used in PVM is that output must first be marshaled and collected into a character array by the user before being output. Support for user-defined objects and formatting manipulator functions is not provided because PVM is implemented in C. The added complexity makes the user program more difficult to port and maintain.

MPI [18] is a standard message passing interface for parallel programming. The current specification of MPI does not require that all processes provide I/O, nor does it specify how the `stdin` or `stdout` of a process is linked to a particular file or device [18] (pp. 287-289). The MPI-2 standards document [6] has addressed this with a chapter on I/O which describes support for collective buffering (shared file access with synchronization) for remotely executing tasks. Non-collective operations are also supported but require additional user synchronization to enforce the desired I/O ordering. The MPI standard supports C and Fortran77 interfaces. Neither of these interfaces supports the use of arbitrary objects or manipulator functions.

The primary problem with the support provided for distributed I/O in PVM and

MPI is that neither provides an easy to use and extensible interface that transparently supports I/O on user-defined objects or manipulator formatting functions. The user is required to provide extra synchronization, buffering, and marshaling of data which increases the programs complexity. The `piostream` library provides high-level object-oriented constructs that transparently support synchronized access to I/O devices on the main host for remotely executing tasks. Data origin identification and synchronized data interleaving are transparently provided by the `piostream` run-time system. The high-level object-oriented interface provided by the `piostream` library supports I/O on user-defined objects through operator overloading, and the use of both `iostream` and user-defined manipulator functions.

Parallel file I/O often involves multiple objects simultaneously reading data or simultaneously writing data to a single file. Sun Microsystem's NFS (Network File System) [17, 20] provides transparent access to remote file systems as if they were local by mounting a remote file system so that it appears as part of the local file system directory structure. The NFS works in a heterogeneous environment of operating systems and networks. The NFS architecture uses a client (a machine that accesses server resources over the network) and a server (a machine that serves resources to the network). A machine can be both a client and a server under the NFS architecture. An NFS server is stateless so that the server does not require information concerning its clients, transactions previously completed, or which files have been accessed. Each client transparently provides the server with a file identifier, a starting byte address, and the length of the transaction in bytes.

In the case of multiple tasks simultaneously reading from the same file, the NFS is an excellent solution if the file systems on which the data resides can be mounted. In some cases however, file systems cannot be mounted for security or administrative reasons. If the file system can be mounted, the data distribution between clients is a concern. For example, striped data distribution requires the reading tasks to share a file pointer so that each client input request reads different tokens of data. However, the statelessness of the server in the NFS results in no easy method of sharing the file pointer. Because the clients must maintain any access information, such as the byte address of the file-pointer, sharing the file pointer between tasks requires explicit synchronization and communication. The extra synchronization and coordination complicates the user code which lowers the portability and maintainability of the program.

The case of multiple tasks simultaneously writing data to the same file is handled very simply by NFS. File writes are not coordinated because the server is stateless (and therefore has no knowledge of previous file write operations). Hence the server cannot eliminate conflicting output of different clients. Explicit user synchronization and communication between tasks performing output is hence required in order to synchronously interleave the file output as desired. The extra synchronization and communication reduces the portability and maintainability of the user program. The proposed `pfstream` constructs transparently provide synchronized access to distributed files for remote tasks so that simultaneous output is synchronously interleaved and not overwritten.

The Condor High Throughput Computing System [13] is a software system designed to utilize unused CPU cycles in a network of workstations. Condor accomplishes this by monitoring workstation activity and running programs on idle workstations. Upon detection of activity from a participating workstation (e.g., a user resumes working on that machine's console), the Condor program is check-pointed and migrated to another idle workstation. Using this approach, a program is more likely to finish computation faster than using only the originating machine (which may be busy). In order to fully utilize Condor's potential, the program must be relinked with the Condor library. No actual code modifications to the user program are required. To allow use of program I/O, the Condor system uses the Remote Unix (RU) facilities [12] which supports Remote Procedure Calls (RPCs). The use of RPCs allows remote machines (which may not grant access to the machine's file system) to access the originating machine's file system for I/O purposes. The main host `stdin`, `stdout`, and `stderr` are also accessible from tasks executing on the remote machine using RPCs.

RU provides many of the same facilities as the `piostream` library for distributed I/O. RU does this through the use of a special C run-time library which provides many system call stubs that allow system calls on the remote machine to be intercepted, redirected, and executed on the main host. RU start-up code opens files on the originating machine that simulate `stdin`, `stdout`, and `stderr` for the program executing on the remote machine. By intercepting file system calls, RU allows the remote process to transparently access files on the main host's file system as if the process were local to that host. RU is a powerful facility for distributed computation, but does have limitations. RU does not support signals or interprocess communication which eliminates synchronization between distributed objects. Although [12] mentions the support of multi-process programs is a possible future direction for RU, neither the current implementation of RU or Condor (which is built on top of RU) appears to support this. The `piostream` library could conceivably be implemented on top of RU to take advantage of RU's RPC facilities. However, RU does not provide support for the synchronized access to the main host I/O devices (because RU is intended for sequential programs). Hence the current implementation of RU is not suitable for implementation of the `piostream` prototype. The `piostream` library transparently handles access synchronization for `stdin`, `stdout`, `stderr` and files on the main host for remotely executing tasks.

Gotwals *et al.* [9], explore the distributed I/O problem by implementing the `d/stream` construct in the parallel language `pC++` [9]. The `d/stream` construct is a language-independent abstraction that supports a number of simple primitives which allow I/O to be performed on distributed arrays with arbitrary object elements. Conceptually, a `d/stream` is a buffer that is used for intermediate storage between the user and a file. Using a `d/stream`, a user is able to insert data into the buffer, then write it to a file at a later time; or conversely read data from the file into the buffer, from which it can then be extracted into a distributed array.

The `d/stream` construct is implemented in the parallel language `pC++`. The implemented construct is called `pC++/stream`. Distributed object support is provided through the use of a compiler dependent feature called *collections*. A collection is defined as a distributed array of objects with additional underlying infrastructure that provides support for arbitrary distributed data structures such as trees. A constraint is enforced that all data read must correspond exactly to data that was previously written by the `pC++/stream` library. More specifically, each read and extract operation must correspond in exact order to a previous write and insert operation. By assigning distributed objects to the same collection, the objects can be output contiguously to the same file, regardless of their address in memory. Entire objects or individual elements can be output contiguously, regardless of which node the data originated from. The `pC++/stream` construct allows the interleaving of individual array elements of multiple distributed arrays that are both inserted into the `pC++/stream` before it is written to file.

Gotwals *et al.* duplicate portions of the C++ `iostream` interface by supporting the use of the `operator<<()` and `operator>>()` methods which can be extended to support user-defined objects. However, some limitations are placed on the user. For example, the interface lacks support for the chaining of I/O method invocations (the use of several input or output operations in the same C++ statement), there is no support for manipulator functions, and a special explicit write call must be made by the user in order to flush the output buffer to a file. `pC++/streams`, the implementation of `d/streams` in the language pC++, is also described for file I/O only, although the extension of the implementation to support the standard I/O devices would likely be possible. As a result of these limitations `pC++/streams` deviates significantly from C++ `iostream` interface. An additional drawback is that `pC++/streams` is based on a compiler dependent construct, collections, which limits its use. Moreover the `pC++/stream` construct requires the use of a parallel file system for data buffering and transmission in a distributed environment.

The `piostream` library conversely is based on the interface of `iostream` which means that chained I/O and manipulator functions are both supported. The `piostream` library does not rely on any language extensions and therefore, depends only on the use of a standard C++ compiler. Lastly the `piostream` library file constructs allow remote objects to perform I/O on the main host file system without dependence on the use of a parallel file system. The `piostream` constructs also do not require input files to have been previously written by `piostream` file I/O operations. Any data file can be used by the `piostream` file constructs for file I/O purposes which increases the portability of the data files and the usefulness of the `piostream` library.

2.3 Active Object Model

The current `piostream` library is implemented using the concept of an *active object*, although any system supporting multiple threads of control, message passing and synchronization could be used. A brief description of the active object model of concurrency is provided to familiarize the reader with the implementation environment. Active objects are the product of combining the object-oriented paradigm, which is concerned with *objects*, and *active* messages [1, 22]. Active messages allow communication and computation to be overlapped, reducing network latency, through asynchronous communication. Each active object has its own thread of control and can be created on any processor. An active object has two main components: a body that defines the active object's activity including acceptance of RMI (Remote Method Invocation) requests; and a message queue that is responsible for storing RMI requests until they are accepted by the active object. The term *acceptance* is used to describe an active object's acknowledgment and execution of an RMI request. Active objects can request RMIs of other active objects synchronously or asynchronously. The requesting object, in both cases, blocks until the RMI arguments are packed and delivered by the run-time system to the destination active object. If the RMI is asynchronous, the active object continues executing; otherwise the active object remains blocked until the RMI is accepted, and the request is de-queued, processed, and a result returned. The remainder of this section describes the active object components, and discusses the issues of active object creation and communication.

2.3.1 Active Object Body

The body of an active object provides its definition for the thread of control. The body is conceptually a `main()` method that executes remotely. The body is responsible for:

- Defining the active object's computation.
- Creating other active objects.
- Making RMI requests to other active objects.
- Accepting specific RMI requests from other active objects.

An active object accepts another active object's RMI request by de-queuing the RMI from the accepting active object's message queue. The active object message queue is described in the following section.

2.3.2 Active Object Message Queue

Active objects allow inter-object interaction through the use of RMIs whose acceptance is controlled by each active object's body of control. Each active object has a message queue, or mailbox, which holds all outstanding RMI requests. When an RMI request is received it is placed into the message queue by the run-time system. Each RMI can later be accepted, removed from the queue, and processed using a variety of criteria. RMIs can be accepted on the basis of: RMI type, RMI originator, or in a FCFS (First-Come First-Served) fashion. The acceptance techniques supported depend on the implementation of the run-time system. The ABC++ implementation environment supports selection of RMIs for acceptance based on RMI type in a FCFS fashion.

2.3.3 Active Object Creation and Lifetime

Active object creation can be done by any existing active object, as well as the original thread of control, the `main()` function (called the main program). The active objects form a hierarchy, with the main program acting as the root node. The properties of the hierarchy are described in the following section.

2.3.3.1 Active Object Hierarchy

Several properties of the active object hierarchy are depicted in Figure 2.5. The main program can create one or more active objects as children. Any existing active object can create one or more new active objects, that are considered children in the active object hierarchy. The lifetime of an active object is defined by its body and is independent of both its parent and children (e.g., an active object can exist beyond its parent's lifetime). Each active object is identified by a run-time system technique (which depends on the implementation) rather than through its parent or its place in the hierarchy. The hierarchy is merely a useful concept for describing the relationship involved in the construction and destruction of active objects. The hierarchy is also useful for describing possible I/O data flow models based on an active object's relationships with its children and its parent.

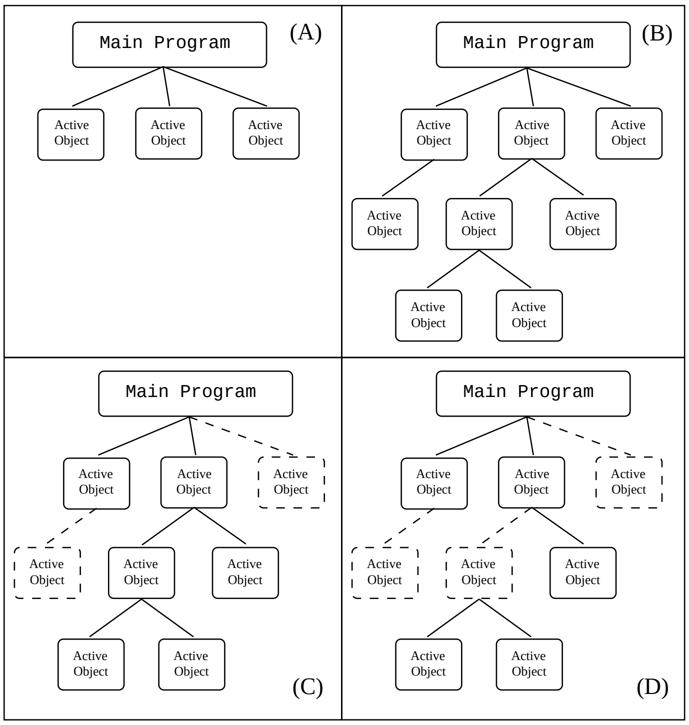

Figure 2.5: This illustration shows the characteristics of the active object hierarchy. (A) shows a main program that has created three active object children. (B) illustrates each active object's ability to create one or more new active objects. (C) shows two destroyed active objects, drawn with a dashed line. (D) illustrates the fact that an active object can outlive its parent. The main program exists until all active objects have terminated.

2.3.3.2 Main Program

The main program is the original thread of control. The main program can create active objects and remotely invoke their methods. However, the main program is not an active object. The main program exists until *all* active objects have terminated and is responsible for starting and shutting down the implementation run-time system.

2.3.4 Active Object Communication

Active objects may directly interact with other active objects through RMI requests. In order to invoke an RMI on another active object, the invoking active object must possess specific information about the remote object. This information includes a handle or pointer, the method to invoke and any method parameters required. Two different types of RMIs are possible — synchronous and asynchronous. A brief discussion of synchronous and asynchronous RMIs is provided because of their effect on a program's performance.

2.3.4.1 Synchronous RMI

Synchronous RMIs require the caller to block until the remote method invocation is accepted, executed, and a result returned. The main benefit of synchronous messaging is that it allows synchronization between remote active objects. Synchronous RMIs can lower performance because the caller blocks until the RMI is completed. Many active objects can potentially end up blocked awaiting RMIs to complete execution. Consequently the design of the `piostream` library should minimize the amount of synchronous RMIs used for inter-object communication.

2.3.4.2 Asynchronous RMI

Asynchronous RMIs do not require the caller to block while the method is being executed by the remote host. They do however, require the caller to block until the RMI parameters are sent to the remote object, after which the caller continues execution. The benefit of asynchronous RMI is the increased parallelism gained from the limited blocking of the calling active objects thus permitting computation to be more effectively overlapped. The drawback of asynchronous RMI is that its implementation may be more difficult due to the extra communication and computation required to handle the return value of an asynchronous RMI. The return value is complicated to handle because its value must be inserted into an active object's address space while the active object is executing. The run-time system must also be able to intercept any access of the return value before the result has been returned. A special type of object, a *future*, is used to hold the result of an asynchronous RMI.

2.3.4.3 Futures

Futures are used as a container for data objects that are returned asynchronously. The future object is used to inform the run-time system that a value will be returned in the future. Upon instantiation of the future object, the run-time system marks the appropriate future as pending. When the future's value is returned, the run-time system marks the future as resolved. If an active object uses a future while that future is still pending, the active object blocks until the future is returned, at which point the active object is unblocked and continues executing. The active object proceeds as normal upon accessing a resolved future.

2.4 ABC++

This section describes the ABC++ implementation of the active object model of concurrency. The `piostream` library design must be modified to meet any limitations of the implementation environment for the development of the prototype. ABC++ is a class library for parallel programming in C++ [1, 2]. It promotes code reuse through the abstraction, encapsulation, and polymorphism properties of the Object-Oriented Programming (OOP) paradigm [14]. ABC++ is written in C++ and requires no preprocessing, compiler or language extensions. The library is portable and object-oriented with a concurrency model based on active objects. It presently runs on SUN workstations, IBM RISC System/6000 workstations and the IBM SP supercomputers. All active object classes must inherit from the base class `Pabc` in order to ensure each active object has the appropriate underlying run-time information. ABC++ supports both synchronous and asynchronous object interactions through RMIs on both distributed and shared memory platforms. ABC++ encapsulates the work required to control threads and synchronize objects thus allowing the user to concentrate on the semantics of the program. The ABC++ library does not provide parallel I/O constructs [16]. The library is built on top of MPI and uses the MPI infrastructure to link each remote active object's `stdout` and `stderr` to the main host window output device. The start-up and shut-down of the library is invoked explicitly by the user in the main program, with the functions `Pinit()` and `Pexit()`. The remainder of this section discusses the components and functions of the ABC++ run-time system.

2.4.1 Body Definition

Each active object class inherits a body from the base class `Pabc`, provided by the ABC++ class library. The default definition of this body repeatedly accepts any RMIs requested by other active objects. This action can be overridden by providing a new `main()` method. This new body can then perform computations or create other active objects. The body is also used to initiate communication by performing RMIs on other active objects or by accepting RMI requests from other objects through its message queue.

2.4.2 Message Queue

Arriving RMI requests are placed in the active object's message queue in the order of arrival. These RMIs can be accepted through two different methods, `Paccept()`, and `Paccept_any()`. `Paccept()` accepts exactly one RMI, from a list of specified methods that the active object is currently willing to accept. `Paccept_any()` accepts the first available RMI. The different ways of accepting RMIs are illustrated in Figure 2.6.

The figure also shows the corresponding RMI requests that would be selected from

```
                   1     2     3    4     5    6
Message Queue [ slap, kick, slap, slap, run, hide]
```

Methods Accepted	**Method Chosen**
Paccept_any();	Slap - 1
Paccept(run);	Run - 5
Paccept(slap);	Slap - 1
Paccept(kick, run, hide);	Kick - 2

Figure 2.6: This diagram illustrates various ways to accept RMIs from active objects. The active object message queue is shown, with the oldest RMI on the left. Messages accepted are on a FCFS (first-come first-served) basis of available and *acceptable* RMIs.

the shown message queue. Both `Paccept()` and `Paccept_any()` accept RMI requests on a FCFS basis. ABC++ does not provide the ability to accept an RMI based specifically on the identity of the RMI requester [16].

If no suitable RMIs are available in the message queue `Paccept()` and `Paccept_any()` block the active object until a suitable RMI request is made. This blocking acceptance limits the design of the `piostream` library. Fortunately ABC++ also provides two methods, `Ppar_accept()` and `Ppar_accept_any()`, that allow the active object to continue executing if no acceptable RMI requests are available. These two methods perform a non-blocking query on the message queue and return a `boolean` result. The method `Ppar_accept()` allows a query for one or more specified RMIs in the message queue. Similarly, the method `Ppar_accept_any()` checks if the message queue contains at least one RMI and returns a `boolean` result.

2.4.3 Active Object Creation

The creation of an active object in ABC++ is a two step process. The user first instantiates a `Pabc_pointer` template class that acts as a handle for the new active object. The user then gives the new active object its own thread of control on the specified host, by invoking the `Pabc_create()` method with the `Pabc_pointer` as an argument, along with any constructor arguments. The host can be allocated by specifying a particular host or a processor allocation technique. The processor allocation techniques include random or round-robin with all hosts, only remote hosts, or only the main host. The default technique is round-robin using all available hosts in the host allocation pool.

The process of creating an active object in ABC++ is illustrated in Figure 2.7. Figure 2.7A provides a generic function prototype for creating a `Pabc_pointer` object

(A)

```
Pabc_pointer<Active_Object_Type> Pointer_name;
Pabc_create(Pointer_name, arg1, arg2);
```

(B)

```
class stooge :public Pabc {
 •••
  // Constructor
  stooge(int strength,
         double speed){...}
 •••
};
```

(C)

```
int arg1;
double arg2;

// Create Smart Pointer
Pabc_pointer<stooge> moe;
Pabc_pointer<stooge> curly;

// Create thread of control
Pabc_create(moe, strength, speed);
Pabc_create(Pproc::local, curly, strength, speed);
```

Figure 2.7: This diagram illustrates the steps required to create an active object using ABC++. (A) provides the general function prototype for creating an active object. (B) defines a simple active object class. (C) shows the code used to create two active objects. The first `Pabc_create()` invocation uses the default round-robin processor allocation technique. The second invocation specifies that the active object should be created on the main host.

and for using the `Pabc_create()` method. Figure 2.7B defines a simple class that takes two arguments for its constructor. Figure 2.7C provides a segment of code that illustrates the use of the `Pabc_pointer` object and the `Pabc_create()` method.

A `Pabc_pointer` is an aggregate class that consists of a *virtual processor id*, an address in memory and a flag that denotes whether the pointer is currently bound or not. The virtual processor id specifies the processor the active object is running on. The memory address specifies where in memory the active object resides.

2.4.4 Communication

ABC++ provides both synchronous and asynchronous RMIs. The use of the ABC++ RMI facilities is illustrated in Figure 2.8. The function prototypes for the communication oriented methods are shown in Figure 2.8A. Figure 2.8B provides a simple active object class definition with two available methods. The body of another active object that invokes remote methods on the simple class is shown in Figure 2.8C. The methods `Pvoid()` and `Pvalue()` support synchronous RMIs with zero and one return value, respectively. Similarly, the methods `Ppar_void()` and `Ppar_value()` support asynchronous RMIs with zero and one return value. The value returned from an asynchronous RMI must be stored in a future[2], which is supported by the template class `Pfuture`.

[2]Discussed previously in Section 2.3.4.3

```
Pvoid(Pabc_pointer, Active_Object::Active_Object_Method, arg1);
result = Pvalue(Pabc_pointer, Active_Object::Active_Object_Method);
Ppar_void(Pabc_pointer, Active_Object::Active_Object_Method, arg1);
Pfuture<int> Object;
Object = Ppar_value(Pabc_pointer,
                 Active_Object::Active Object Method, arg1, arg2);
```
(A)

```
class stooge :public Pabc {
  •••
  // Constructor
  stooge(int strength,
         double speed){...}
  // Methods
  void slap(int hand){
  {
    pain = hand * strength;
  }
  double run(double time)
  {
    return speed * time;
  }
  •••
};
```
(B)

```
int strength, hand, arm_length;
double speed, distance;

// Create Smart Pointer
Pabc_pointer<stooge> moe;

// Create thread of control
Pabc_create(moe, strength, speed);

// Return variables
double distance;
Pfuture<double> long_distance;

// Synchronous messaging
distance = Pvalue(moe,
       stooge::run, time);
Pvoid(moe, stooge::slap, hand);

// Asynchronous messaging
long_distance = Ppar_value(moe,
       stooge::run, time);

 •••

if (long_distance < arm_length) {
    Ppar_void(moe, stooge::slap, hand);
}
```
(C)

Figure 2.8: This diagram shows synchronous and asynchronous RMIs using ABC++. (A) provides the general function prototypes of the synchronous RMI methods, `Pvalue()` and `Pvoid()`, and the asynchronous RMI methods, `Ppar_value()` and `Ppar_void()`. (B) defines a simple active object class with two methods. (C) shows program code that uses asynchronous and synchronous RMIs with the active object defined in (B). The `Pfuture` class is also shown in (C) with the asynchronous RMI method `Ppar_value()`.

Chapter 3

Postream

Our `piostream` library is designed to solve several problems encountered when using the `iostream` library in parallel and distributed environments. The central problem is that using standard techniques for creating threads of execution on remote hosts do not properly utilize the proper I/O devices. Each thread's `stdout` is linked to what is essentially an incorrect device for the purpose of performing output on the main host. If remote threads are created using an `rexec` system call, a socket is created and given to the `stdout` of the thread on the remote host (which is clearly not the correct device). Thus, using `cout` would require the programmer to add special code to their program to coordinate with the main host. The design of the `postream` construct alleviates the programmer's burden of writing special code for performing I/O in such environments. In addition, the `postream` library also solves the problems of unsynchronized data interleaving (which can occur when access to the output stream is not properly synchronized) and data origin identification (which means that the source of the output is not identified).

This chapter describes the design and implementation of the high-level object-oriented `postream` construct. The chapter describes the problems of unsynchronized data interleaving and data origin identification which occur when multiple objects produce output using the same output stream. Several `postream` designs that were considered are presented and discussed. The `postream` client-server model used for the prototype implementation is presented along with an analysis of related design and implementation issues. An overview of the `postream` library prototype architecture is provided. The modifications required to the `postream` library to support active object output to `stderr` on the main host with synchronized data interleaving and data origin identification are described at the end of the chapter.

3.1 Unsynchronized Data Interleaving

The problem of unsynchronized data interleaving is prevalent in parallel and distributed programs that use the C++ `iostream` library. The C++ `iostream` library is reentrant but places limitations on how multiple threads interact with output streams [19]. In order to ensure that the code is reentrant, the `iostream` objects written for a multi-threaded environment are synchronized within the scope of the public member functions of the `cout` object. In other words, each individual function call is atomic with respect to access of the output stream. However, the interface of `cout` allows for multiple invocations of `operator<<()` in a single output statement (as shown in Figure 1.2). Consequently a chained output statement is not performed atomically — only individual `operator<<()` invocations are atomic. Hence, when multiple active objects perform output to the same stream using chained invocations, the resulting output is in arbitrary order.

To illustrate the problem of unsynchronized data interleaving, consider Figure 3.1 which describes a simple class whose objects are active. The active object's body uses

```
class active_object
{
public:
   // Constructor
   active_object(int i) :input(i), result(0){ }
   main()
   {
     cout << "Input of " << input << " received" << endl;
     result = get_result(input);
     cout << "Result = " << result << endl;
   }

private:
   int result;
   int input;

   // Compute integer result
   int get_result(int input){...}
};
```

Figure 3.1: This diagram defines an active object body that uses multiple chained output statements.

an input parameter and calls the method `get_result()`, which returns an integer. The object then outputs this result using the `ostream` object `cout`. The output resulting from three of these active objects executing simultaneously is shown in Figure 3.2. The output in Figure 3.2A is difficult to interpret as the output values are interleaved and merged together. The same output is shown in Figure 3.2B with synchronized

(A)

```
Input of Input of 388 received receivedInput of

16 receivedResult = 4Result =

Result = 198
```

(B)

```
Input of 8 received
Input of 38 received
Input of 16 received
Result = 4
Result = 8
Result = 19
```

(C)

```
<curly:7890:1> Input of 8 received
<moe:1234:2>   Input of 38 received
<larry:5678:4> Input of 16 received
<curly:7890:1> Result = 4
<larry:5678:4> Result = 8
<moe:1234:2>   Result = 19
```

Figure 3.2: This diagram shows output from three concurrent active objects executing code from Figure 3.1. (A) illustrates output with unsynchronized access to the output stream. The resulting output suffers from the unsynchronized data interleaving problem. (B) shows the same output with synchronized access to the output stream. The output still suffers from the problem that its origin is not identified. (C) shows the same output with synchronized interleaving and data origin identification consisting of the host name, the process id, and the thread id. The data origin identification width can be set using a `postream` manipulator function.

interleaving. The output still suffers from the data origin identification problem. The output shown in Figure 3.2C uses synchronized interleaving and data origin identification consisting of the process id, thread id, and the host name. The example shown is trivial because of the small number of output statements and concurrent objects. However, unsynchronized data interleaving problem can be much more serious in the case of a parallel program with large numbers of concurrent objects each printing complicated debugging output.

One solution, and the most common solution, to the unsynchronized data interleaving problem is to force the user to coordinate the access of the output stream through a synchronization mechanism such as a lock or a monitor. This further complicates user code and can be misused because the user is responsible for acquiring and releasing the lock for each output operation. Hence, the use of a lock or monitor at the user-level is not an acceptable solution. The `postream` library transparently synchronizes access to the output stream thus greatly simplifying the user code. The `postream` construct permits data to be interleaved using a suitable granularity such as the flush of the output stream as shown in Figure 3.2B. The flush of the output stream is handled by the manipulator functions `endl` and `flush` through the `postream` `pout` object.

3.2 Data Origin Identification

Re-examining Figure 3.2B, the output still suffers from the problem that its origin is not identified, which makes program debugging and execution more difficult. From the output alone, one cannot determine which active object produced which result. Several possible user-level solutions to this problem exist. One solution is to redirect each active object's output to a separate output window[1]. If the parallel system does not support output redirection to separate windows, the user can explicitly provide the necessary mechanisms. User-level output redirection however, adds significant complexity to the user code which decreases both the portability and maintainability of the program. Another problem with using separate output windows is that it does not allow for the output from different hosts to be examined simultaneously. Sometimes it is useful to examine the output together for debugging purposes such as determining the ordering between output from different active objects.

A second solution is for the user to add an identification system on the objects that would subsequently be prepended to each individual output statement (e.g., each flush of the output stream). This solution adds complexity to the user code and again makes the program more difficult to port and maintain.

In the `postream` library, unique identification tags are transparently generated by the run-time system and prepended to each sequence of output data as shown in Figure 3.2C. The `postream` identification tag consists of three variables: the host name, the process id, and the thread id. The identification tags are created for each active object by the run-time system. A library that provides data origin identification and synchronized data interleaving is invaluable for writing and debugging parallel and distributed programs. Furthermore, such a library contributes to the simplicity of user code which lowers the cost of code maintenance.

3.3 Postream Architecture

Three different models were considered to solve the problem of data interleaving. These models conceptualize different approaches of handling each active object's output data. The first model, called pass to parent, handles data interleaving by forcing each active object to be responsible for its heir's output. Ultimately, the main program is responsible for all output data and therefore controls the interleaving of the output data. The second model, called shared lock, synchronizes access to the output stream using a shared lock. This model requires each active object to have existing access to `stdout` on the main host. The third model, called client-server, uses a dedi-

[1] Treadmarks [11], a popular distributed shared memory system, supports separate window output.

cated active object which executes on the main host to synchronize output to `stdout`. We have selected the client-server model for the `postream` design and demonstrate its feasibility by using it to implement the `postream` prototype. Sections 3.3.1-3.3.3 discuss each model by first describing the model and then examining the model with respect to a number of design issues.

3.3.1 Pass to Parent Model

The pass to parent model requires all output data to be passed from the active object performing output to the parent of that object. Each active object has a parent, which is either the main program or an active object in the system. The ancestral relationship of the active objects forms a hierarchy (as described in Section 2.3). Output data is transmitted up the hierarchy toward the root object. Therefore, each active object must maintain the address of its parent object which increases coupling between the objects. Each active object must receive output from its children and pass the output to its parent. Ultimately the main program is responsible for handling all output because it is the ancestor of all active objects. Therefore, the main program can control data interleaving by synchronizing the output of data that it receives.

Figure 3.3 depicts a hierarchy with the direction of data movement indicated with respect to each active object. The normal active object hierarchy is shown in Figure 3.3A. The main program is the ancestor of all active objects and is responsible for synchronizing the output of data to `stdout` on the main host. The flaw with this model is illustrated in Figure 3.3B. Recall from Section 2.3.3 that active objects can enter and leave the system at any time. Subsequently, active objects that are children can outlive their parents. The termination of a parent active object before its child, results in a missing link in the active object hierarchy (as seen in Figure 3.3B). Hence the child's output data cannot be transmitted up the disconnected hierarchy to the main program. Dynamic maintenance of the hierarchy would solve this problem but would present substantial implementation problems in order to properly maintain and reorder the hierarchy during program execution. The pass to parent model was rejected because it does not adequately handle a disconnected hierarchy of active objects.

3.3.2 Shared Lock Model

The shared lock model uses a shared lock to synchronize access to `stdout` on the main host. This model assumes that active objects executing remotely have existing access to `stdout` on the main host. The implementation environment, ABC++, supports direct access to `stdout` and `stderr` for all active objects using the underlying MPI infrastructure. The shared lock can be implemented using an active object server

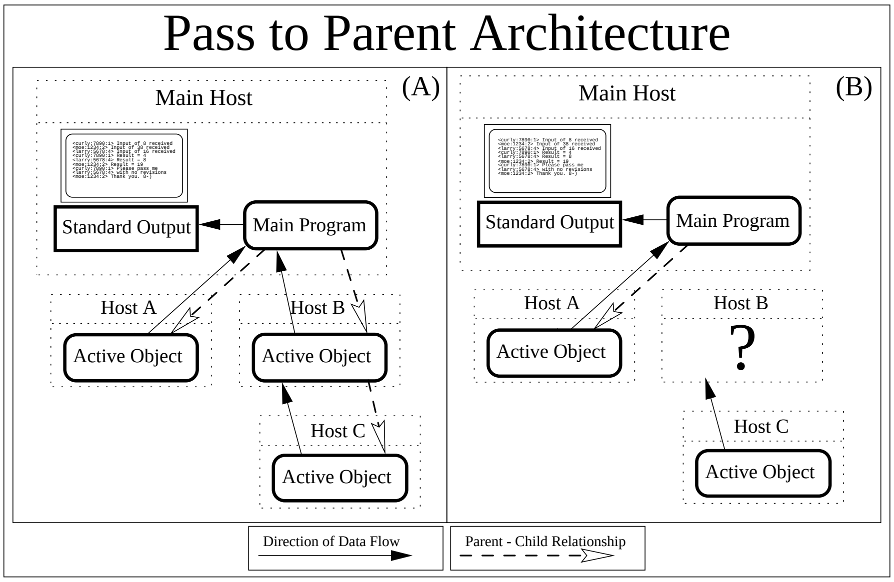

Figure 3.3: This diagram illustrates the directional data flow and active object hierarchy in the pass to parent model. (A) depicts a normal hierarchy in which each active object is responsible for its childrens' output. The main program is the ancestor of all active objects and hence is responsible for outputting each active object's data to `stdout` on the main host. (B) illustrates the flaw in the pass to parent model. The termination of a parent object can result in no clear path from its children to the main program.

or a shared object[2]. The communication between the shared lock and active objects performing output is depicted in Figure 3.4. Before any active object can output data to `stdout` on the main host, the active object must make an acquire lock request. When the lock acquire has been granted, the active object may directly output to `stdout` on the main host. When the active object completes the output operation, the lock is released. All lock acquire and release requests are performed transparently by the run-time system.

Two concerns in the shared lock model include when to acquire and when to release the shared lock. The desired level of data interleaving was defined in Section 3.1 to be around the flush of the output stream. Consequently it is logical to release the

[2]A shared object is implemented through a DSM (distributed shared memory) run-time system. A DSM system allows an object to exist in multiple address spaces so that the object appears local to different active objects. The DSM run-time system is responsible for maintaining the consistency of the object's data between each address space.

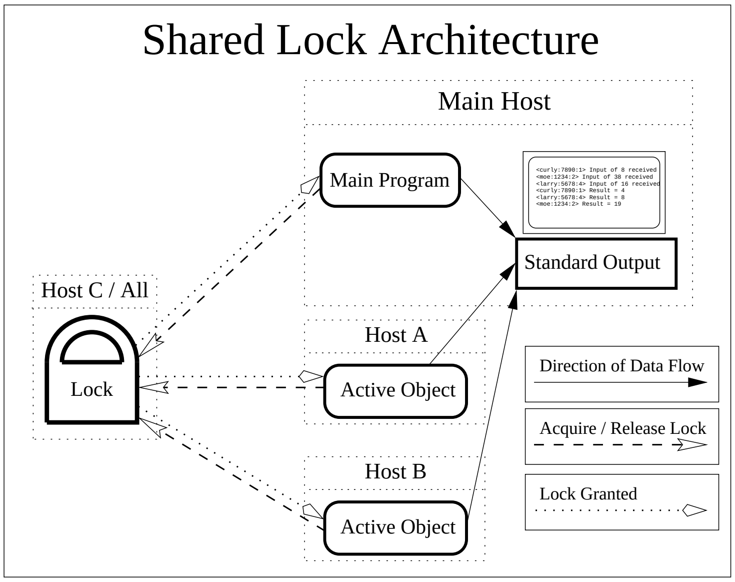

Figure 3.4: This diagram illustrates the directional data flow and communication between the shared lock and active objects performing output in the shared lock model. The shared lock could be implemented using either a shared object or an active object and hence exists on one host or is shared between each active object's address space. This model assumes each active object has direct access to `stdout` on the main host. Each active object may only output data to `stdout` on the main host after acquiring the lock. The active object is responsible for releasing the lock after performing output.

lock within the flush manipulator functions (`flush` and `endl`) of the output stream. The decision of when to acquire the lock is not as simple. Two choices are evident:

- Obtain the lock when the active object flushes the output stream.
- Obtain the lock any time `operator<<()` is invoked and the active object does not already possess the lock.

The different acquisition approaches are illustrated in Figure 3.5.

Approach 1 requires the active object performing output to buffer data, because an arbitrary number of `operator<<()` invocations could be performed (each outputting data) before the output stream is flushed. For example in Figure 3.5, `obj1`, `obj2`, and `obj3` must all be buffered before being output.

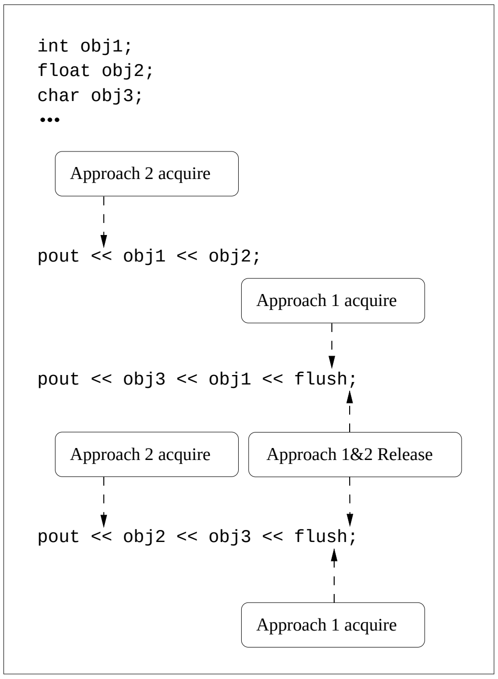

Figure 3.5: This diagram shows the different approaches of acquiring and releasing the shared lock on the output stream. Approach 1 acquires the lock during any `operator<<()` invocation, if it does not already possess the lock, and releases it on the flush of the output stream. Approach 2 acquires *and* releases the lock when the output stream is flushed.

Approach 2 is easier to implement because it requires no buffering. The lock is acquired at the start of each output sequence and released when the stream is flushed. Each token of data is then inserted directly into the output stream on the main host. However, approach 2 is not very robust or efficient. If the user neglects to flush the output stream (and therefore does not release the lock) *any* other active object that outputs data, and subsequently attempts to acquire the lock, will block indefinitely. Even if the user flushes the stream properly, the synchronous nature of lock acquisition impacts performance because the lock is held for a longer duration of time under approach 2. Hence, approach 1 is a better lock acquisition and release scheme for robustness and performance reasons.

3.3.3 Client-Server Model

The client-server model uses an active object as a dedicated output server to control the synchronization of active object output. The server executes on the main host and hence has access to `stdout` on the main host. Each active object that produces output is a client of the output server, including the main program. The coordination and transfer of data required between each client and the output server is handled transparently by the run-time system.

The direction of data flow and communication between the client and server is illustrated in Figure 3.6. Active objects, which may execute remotely, transmit output

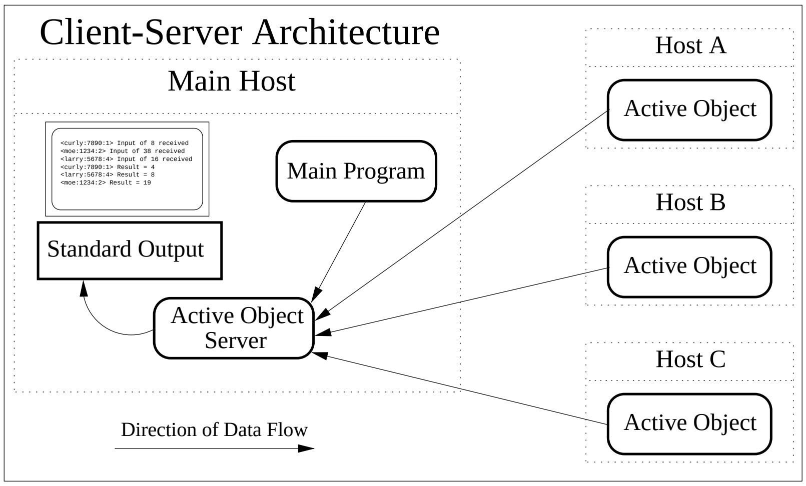

Figure 3.6: This diagram illustrates the directional data flow in the `postream` client-server model. A dedicated active object server resides on the main host and accepts output from active objects that may be executing remotely. The server interleaves the output and prepends each client active object's data origin identification consisting of the process id, thread id, and host name.

to the server. The server controls the data interleaving of the output and prepends each active object's output with the appropriate data origin identifiers. The data origin identification is transferred during the initial communication between the client and server, afterwhich the server stores the origin information. All data transfers between the client and server are performed asynchronously which improves the performance of the client-server model over the synchronous communication required in the shared lock model. The performance improvement is because asynchronous RMIs increase the parallelism of the active objects performing output compared to the use of synchronous RMIs. Although either the shared-lock model or the client-server model could be used for the `piostream` prototype, the client-server model was chosen because it does not rely on existing access to `stdout` on the main host for all remote active objects. The client-server model is hence more portable and is easier to implement.

The main concern with this model is when to transfer data to the output server. The following represents two possible choices:

- Approach 1) Transfer the data after a flush of the output stream.
- Approach 2) Transfer the data after each invocation of the `operator<<()` function.

The storage and transfer of data using both approaches is illustrated in Figure 3.7 using an example chained output statement.

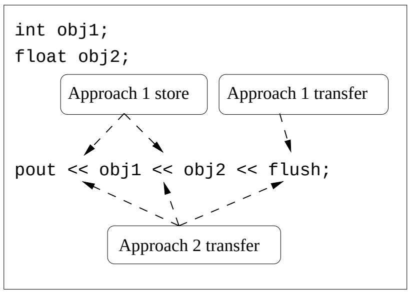

Figure 3.7: This diagram illustrates the different approaches towards transferring output data to the output server. Approach 1 stores all data until a flush of the output stream, at which time the buffer is transferred. Approach 2 transfers each token of data after each invocation of `operator<<()`.

Approach 1 requires each active object to buffer its output until the flush occurs. A limit on the size of the client buffer is enforced in a fashion similar to the `ostream` object[3]. If the client buffer's size limit is reached, the run-time system transparently transmits the output data to the output server using a remote method invocation.

Approach 2 is easier to implement because no client buffering is required. However, approach 2 requires more message transfers than approach 1 because each `operator<<()` invocation transmits data between the client and the output server.

[3]The limit imposed in several `iostream` implementations, including the implementation environment IBM Cset1.1.1.4 for Solaris is 1024 bytes.

Because such messages are usually expensive in a distributed system, approach 1 is more efficient than approach 2 and is therefore used for the design of the `postream` construct.

Using approach 1, if the server receives data from the client because the client's buffer has filled, the output server stores this data in a storage buffer located on the main host. Data received because of the invocation of a flush manipulator is immediately output to `stdout` on the main host. The server maintains a vector of storage buffers, one for each client active object. Each client's buffer on the server is indexed by a vector identification key. The server uses a dynamic sized vector to serve the growing number of clients in the system. The vector starts with a programmer controlled size and is doubled each time the vector becomes full. An index key is used by the server to identify each active object performing output. The key is requested by the client the first time data is transmitted from the client to the server. The key cannot be requested when the `postream` object is instantiated because the message passing facilities of the client active object may not be enabled at that time. The active object is not fully instantiated until the active object's data members, including the `postream` object, are instantiated. The run-time system transmits the client active object's data origin identification to the server object.

Each client's data origin identification is stored with the client's storage buffer on the server. The server then prepends the data origin identification to each client's data when it is output to `stdout`. The storage buffers on the server are also limited in size. If the buffer size is exceeded, the buffer is flushed to `stdout` with the appropriate data origin identification. The handle of the server is inserted by the run-time system into each newly created client active object. The handle cannot be set by the active object constructor because the active object is instantiated as a user object. Hence the active object constructor is invoked by the user and consequently the handle cannot be included as a constructor argument without changing the user interface. The output server is created and terminated by the run-time system within the parallel system start-up and shut-down routines.

As noted in Section 2.1.2, the support of manipulator functions with parameters is complicated because three arguments must be passed in the binary `operator<<()` function. As discussed, the specific manipulator function's identity must be passed to `operator<<()`. Two techniques can be used to represent the manipulator function name: a global function or a class. Both of these approaches result in limiting the extensibility of the `iostream` manipulator functions for parallel and distributed environments.

For example, consider the case of the `setprecision` manipulator. If `setprecision` is implemented as a function, it takes an integer parameter and returns an `ostream` object. Overloading this function for use in a parallel environment results in a function that takes an integer parameter, and returns a `postream` object. The overloaded and original `setprecision()` functions differ only by the type of object returned, but C++ does not permit overloading by return type alone [5]. Hence, the implementation of the `setprecision()` manipulator as a function prevents the extension of `setprecision` because the function cannot be properly overloaded.

Now consider the case where the manipulator function's identity is passed as a class type. The example manipulator function `setprecision` is implemented as a class. This class, which is part of the standard `iostream` manipulator library, can not be modified to add parallel functionality[4]. A second class cannot be defined with the same class name[5] to handle the parallel implementation. Clearly using a global function or a class type with the same name can not be used to provide parallel versions of parameterized manipulators.

To circumvent this problem, the parallel manipulator library (`piomanip.h`) is used in our prototype. This manipulator library is used in lieu of the sequential `iostream` manipulator library (`iomanip.h`). It should be noted that the use of both the `piomanip` and `iomanip` libraries in the same compilation is undefined[6]. This requires the parameterized manipulator function to be used only with the `piostream` library. The `piomanip` library fully reimplements the `iomanip` library but adds distributed support for the manipulator functions that require parameters (as described in Section 2.1.2). The `piomanip` library supports all of the `iomanip` manipulator functions with the same manipulator names and a chained interface.

The `postream` client-server model supports the extensible `iostream` interface that allows the output of user-defined classes through operator overloading, the chaining of `operator<<()` functions, and the use of output manipulator functions. All actions performed on behalf of the clients as related to output are transparently supported by the `postream` library and the underlying run-time system. The code at the user-level simply uses `pout` in the same fashion as `cout` is used.

3.3.4 Architecture Overview

The `postream` client-server architecture has a client and a server component. The server runs on the main host. The client is a data member of each active object and resides where the active object is executing. The major components of the server and client are described in this section. The interface of the client and server can be found in Appendix A.

[4]The required addition would be to declare the overloaded `operator<<()` function as a friend of the `setprecision` class.

[5]The exception to this is the concept of *namespaces* [4, 5]. Namespaces allow the duplication of class names in different scopes. Manipulator functions exist in the same global scope and hence, namespaces cannot be used to solve the problem.

[6]The likely result is a compilation error due to either a type or a method conflicts.

Server Architecture: The server component of the `postream` library is a single class, `postream_server`, which uses several smaller classes for implementation. A single object of this class, `po_server` is instantiated to act as the output server. The server body continually accepts RMIs from clients until it receives a terminate RMI from the run-time system. The `postream_server` class contains a dynamically sized vector of buffers, one per client, for storing output data. If the vector becomes full, it is doubled in size. Each client is assigned a buffer when the run-time system first transfers the client's data to the output server. Each buffer is also dynamically sized with an initial size of zero. Each buffer has a size limit which if reached results in the data in the buffer being flushed to `stdout`. The client's data origin identification information is stored with the client's respective buffer on the server.

When a client transmits data to the server, a `boolean` flag is used to indicate whether more data is expected in the current output sequence. An output sequence is considered to be all of the information between two sequential flushes of a client's output stream. Therefore, the end of any particular sequence is signified by the flush of the client's output stream. Hence, more data would be expected if the transfer of data occurs because of an overflow of the client's local buffer. If the user does not flush the output stream, the data is buffered on the client until an overflow occurs, and the run-time system transfers the data to the server for storage in the appropriate server buffer. The overflow of a server buffer results in the immediate flush of the data to `stdout` on the main host with the appropriate data origin identification. If more data is expected, the data is stored on the server in the client's storage buffer using the client's key as an index in the vector. If no more data is expected, the server outputs the client's data origin identification, the contents of the client buffer on the server and the data transmitted with the flush (which has not been inserted into the server buffer) to `stdout` on the main host. The server maintains another `boolean` flag for each client which is used to suppress or show the data origin identification when the client's data is output. This flag can be modified through a public method using the `postream` manipulator functions `hideid` and `showid`. All `postream_server` methods are used through the `postream` client interface and are invoked by the run-time system on behalf of the client. The `postream_server` supports the following methods:

- *Request key* (`request_Key()`) — this public method allows a client to obtain a unique identification key for communicating with the server. This method is invoked by the run-time system the first time output data is transferred to the server. This exchange can not be performed at object creation time because the active object communication mechanisms may not have been instantiated at this time. This method takes the client's identification information (host name, process id, and thread id) as parameters.

- *Transfer data* (`transfer_Data()`) — this public method is invoked by the run-time system to transfer data to the server for output to `stdout`. The client's index key and output data are provided as arguments. A flag specifying whether the passed data buffer is the last of the current sequence is also provided.

- *Transfer function* (`transfer_Function()`) — this public method is invoked by the run-time system to transfer manipulator function information to the server. All `postream` manipulator functions that are based on the standard `iostream` manipulator functions are invoked on the client side of the `postream` construct using the client buffer. Therefore, the `transfer_Function()` method does not require knowledge of these `iostream` based manipulator functions. The `postream` specific manipulator functions such as `hideid` and `showid`, are supported by the `transfer_Function()` method because they directly affect the `postream_server`'s behaviour. Each `postream` specific manipulator function is identified through the use of an enumerated type. The run-time system provides the index key and manipulator function's enumerated value as arguments.

- *Flush all buffers* (`flush_All()`) — this private method is invoked by the server before terminating. All buffers on the server are flushed in sequence using data interleaving and the appropriate data origin identification.

- *Shut down server* (`terminate()`) — this public method is invoked by the run-time system to shut down the server.

Client Architecture: The client component of the `postream` library consists of one class, `postream`. At first glance, extending the existing `ostream` class through inheritance seems to be the best approach for the design of the `postream` class. However, several problems exist with this approach.

The main problem with extending the `ostream` class through inheritance is the `ostream` construct's dependence on the `operator<<()` friend functions. These friend functions comprise a significant portion of the `ostream` library user interface. The friend `operator<<()` functions cannot be extended through inheritance because they are not member functions of the `ostream` class. Therefore, to support the same interface the `postream` library can either use the existing `ostream operator<<()` friend functions or create new versions by method overloading. Both approaches do not work because:

- Using the existing `ostream operator<<()` functions causes a problem with upcasting when used in chained invocations because the `endl` and `flush` functions are typically used at the end of a chained invocation. The overloaded postream flush and endl functions encapsulate the required facilities for communication with the output server in order to preserve the iostream interface. The upcasting problem is illustrated in Figure 3.8. The existing operator<<()

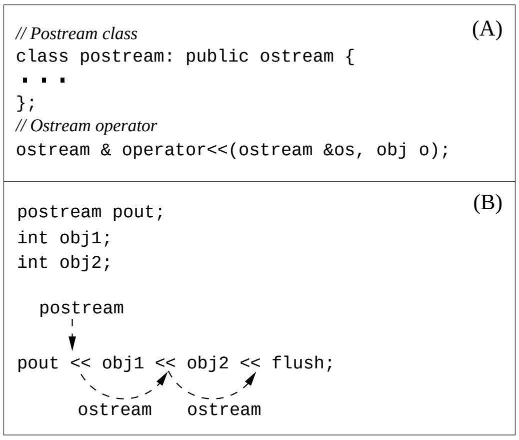

Figure 3.8: This diagram illustrates the problem encountered using the existing ostream operator<<() functions with the derived postream class. (A) shows the derived class postream which publically inherits the base class ostream, including the friendship of the operator<<() functions. In (B), the postream object pout is instantiated. The pout object is used in a chained invocation with two objects and the flush manipulator function. The chained invocation uses the postream object as an argument, but, the object is upcast into an ostream object by the first operator<<() function. The operator<<() function returns the ostream object which is used in subsequent operator<<() function invocations in the chain. The ostream flush manipulator function is invoked at the end of the chain instead of the postream flush manipulator function. The ostream flush manipulator function does not support the required communication with the output server.

function takes an ostream object as an argument. Because postream is inherited from ostream, the postream object is accepted as an argument for the first operator<<() invocation. However, the postream object is upcast into the more general ostream object. This ostream object is returned and used by subsequent operator<<() invocations in the chain. The operator<<() insertion functions still behave properly because the data is inserted into the underlying streambuf because a postream object is an ostream object and therefore inherits the streambuf contained in the ostream class. However, when the flush manipulator function is invoked at the end of the chained invocation the `ostream` version of the `flush` function will be invoked rather than the `postream flush` function. The `ostream flush` and `endl` functions do not support the necessary communication with the output server. Hence, using the existing `ostream operator<<()` functions is not an acceptable solution for implementing the `postream` class in a distributed environment.

- Using overloaded `ostream` and `postream operator<<()` functions leads to the problem that the compiler cannot distinguish between the two functions. Both the existing `ostream` and overloaded `postream` versions of the `operator<<()` functions are shown in Figure 3.9. Both versions have two parameters, an in-

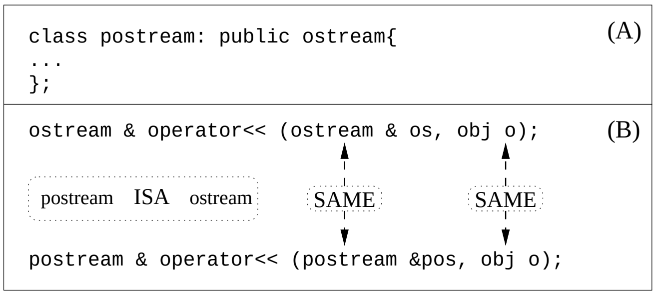

Figure 3.9: This diagram illustrates the problem encountered with using inheritance and overloading `operator<<()` functions. Class `postream` is publically inherited from class `ostream` as shown in (A). Both `operator<<()` functions are shown in (B). Each `operator<<()` function has two parameters: the output argument and an `ostream` or `postream` object respectively. The `postream` object *is* an `ostream` object however and consequently the compiler cannot determine which function to use because the function prototypes are indistinguishable.

  teger and an `ostream` or `postream` object respectively. Recall however, that a `postream` object *is* an `ostream` object because `postream` is publicly inherited from `ostream`. Consequently the compiler cannot distinguish between the overloaded functions and will not compile a program with both function declarations. Therefore overloading the `operator<<()` function is also not an acceptable solution.

The main advantage inheritance provides is code re-use. However, even if the problem of the `ostream operator<<()` interface could be solved, the extension of the `ostream` class is still complex. Due to these complexities, the inheritance approach was rejected and a new `postream` class was implemented that parallels the `ostream` class. This approach is simpler and easier to implement for our prototype.

Each client active object contains an object of the `postream` class named `pout`. This is implemented in the prototype by adding a data member, `pout`, to the root class in ABC++. Since all user active object classes inherit from ABC++'s root class, `Pabc`, every client contains a `pout` object as a data member. This object, `pout`, then acts as an interface between each active object and the `postream_server` object.

Data members of the `postream` class include the index key provided by the `postream_server` object, the handle of the `postream_server` object, and a storage buffer. The handle of the server is used to access the processor id, and the machine name or Internet address of the output server. The `postream` class supports both the output of user-defined objects (through overloading) and manipulator functions.

The interface and behaviour of the `postream` methods are similar to their `ostream` counterparts. The `postream` class supports the following methods:

- *Set address of* `po_server` (`set_POS_Server()`) — this private method is invoked by the run-time system to set the `postream_server` handle which consists of the server run-time host and memory address.

- *Get client identification* (`get_Identification()`) — this private method is invoked by the run-time system to gather the client active object's host name, process id, and thread id for data origin identification. The data origin identification is transmitted to the server by remotely invoking the server method `request_Key()` on the server.

- *Output operator for predefined types* (`operator<<` {`predefined type`}) — this public method allows the client to insert all predefined data types (except character strings) into the local buffer. Character strings are handled separately because of their arbitrary size.

- *Output operator for character strings* (`operator<<` {`character string`}) — this public method is used to insert character strings into the buffer. If the character string is too large to fit into the client buffer, it is divided and transmitted to the `postream_server` by remotely invoking the `postream_server` method `transfer_Data()` on the server.

- *Output operator for manipulator functions* (`operator<<` {`manipulator function`}) — this public method is used to invoke manipulator functions including `endl` and `flush` in a chained fashion. The passed manipulator function is invoked with the `postream` object as an argument.

- *Manipulator functions* (`flush`, `endl`, `hex`, etc.) — these manipulator functions implement the parallel variants of the standard `iostream` manipulator functions. The parallel manipulator functions invoke their standard `iostream` counterparts which set the appropriate format fields on the underlying `postream` client buffer. These manipulator functions are invoked in a similar manner to their sequential counterparts. Many of these functions can also be invoked in chained or unchained fashion as described in Section 2.1.2.

- *Set function* (`setf()`) — this public method is used to set manipulator functions such as `showbase` and `showpos`.

- *Unset function* (`unsetf()`) — this public method is used to unset manipulator functions such as `showbase` and `showpos`.

- *Send data* (`send_Data()`) — this private method encapsulates the transmission of data to the output server by invoking the `postream_server` remote method `transfer_Data()` on the server.

- *Send function* (`send_Function()`) — this private method encapsulates the invocation of `postream` specific manipulator functions on the output server by remotely invoking the `postream_server` method `transfer_Function()` on the server.

- *Show / Hide data origin identification* (`showid/hideid`) — these manipulator functions allow the client to suppress the data origin identification that is prepended to each client's output data. These manipulator functions are invoked by the client in a fashion similar to standard manipulator functions.

3.4 Perr

UNIX provides a second output device for error messages because of the need to distinguish between normal output and error messages. The C++ standard `iostream` library supports this distinction by providing a second `ostream` object `cerr`. Hence, the `piostream` library should also provide a construct that outputs to `stderr` on the main host. Although active objects can use the standard `iostream` object `cerr` to output to `stderr` on the main host, the output will suffer from the problems of data interleaving and data origin identification, described in Sections 3.1-3.2. Hence, by providing a `perr` object, the `piostream` library provides data interleaving and data origin identification for error messages that are output to `stderr` on the main host. The following section will describe the modifications to the `postream` client-server architecture that are required to support output to `stderr` on the main host with synchronized interleaving and data origin identification.

3.4.1 Perr Architecture

The server component of the perr construct is handled by the postream_server class previously described. The postream_server is modified to support output to either the stdout or stderr device on the main host. The same postream_server object is used for both the pout and perr constructs. The postream_server method request_Key() is modified to take an additional parameter, a flag that specifies the appropriate output device, stdout or stderr. The flag is stored with the client's buffer on the server in addition to the client's data origin identification information. The server uses the flag to determine the appropriate output device for each client's output[7]. The output device used by the postream construct is set by the run-time system when the postream object is instantiated. The instantiation of the perr object is handled using the same technique as pout. A perr object of type postream is added as a data member of the active object root class in ABC++.

3.5 Summary

This chapter presented an overview of the design and implementation of the postream component of the piostream library. Active objects that execute on remote hosts often cannot output to stdout on the main host because remote cout and cerr objects may not be connected to meaningful output devices. The user must provide complex buffering and coordination between active objects in order to perform output to stdout or stderr on the main host. If remote active objects can access stdout and stderr on the main host, the user must still provide data origin identification and enforce synchronized data interleaving to ensure that the output is coherent. This adds complexity to the user code which lowers portability and maintainability.

The postream construct provides parallel and distributed output facilities for remotely executing active objects. The postream run-time system transparently provides support for all data transfer, buffering, and synchronized access to stdout and stderr on the main host. Data origin identification and data interleaving is provided by the postream run-time system for each active object performing output. The postream construct provides a similar interface and behaviour as the C++ ostream construct. At the user-level, active objects simply use pout or perr to output data to stdout or stderr respectively. As in C++, operator<<() can be overloaded at the user-level. An example of the postream syntax using the pout and perr objects is shown in Figure 3.10. The postream library is an object-oriented solution that

[7]Note that each active object can represent two clients on the postream_server if output is performed with both pout and perr. Each object has a different buffer and destination, which requires a different index number, and hence each can represent a different client.

```
int mass;
float acceleration;
float force;
•••
if (mass < 0){
  perr << "Negative mass is not permitted" << endl;
  exit(0);
}
pout << "The mass of the object is "
     << setprecision(0) << mass
     << " kg." << endl;
pout << "The object’s acceleration is "
     << setiosflags(showpos) << acceleration
     << " metres per second squared." << endl;
pout << "The force of the object’s impact is " << flush;
force = float(mass) * acceleration;
pout.precision(6);
pout.setf(scientific);
pout << force << " Newtons. << endl;
```

Figure 3.10: This diagram illustrates the syntax used to output with the `postream` `pout` and `perr` objects. Integers, floats, and character strings are output using a chained invocation. Various formatting manipulator functions are used with both direct and chained invocation techniques.

offers significant advantages to programmers for writing and debugging parallel and distributed programs. The `postream` construct simplifies user output operations and hence, increases the portability and maintainability of the user program.

Chapter 4

Pistream

Like sequential programs, many parallel and distributed programs also require input to perform meaningful computations. Although program input is often obtained from files, reading data from the user console is also common for the purposes of user input. A file can also be easily redirected to `stdin` allowing the user to input data from files. In a distributed environment however, the use of `stdin` by active objects residing on remote machines is often not meaningful because `cin` objects on the remote host are frequently not associated with `stdin` on the main host. Little attention has been paid to techniques for providing input from `stdin` on the main host to tasks executing on remote hosts. In many parallel and distributed systems, the use of `stdin` by remote tasks is often either not uniformly supported [18] (pp. 287-289) or is simply not supported at all [8]. Remote Unix allows remote `stdin` access by invoking a shadow process that intercepts remote system calls and redirects them to the program's host of origin. More detail on Remote Unix is provided in Section 2.2. Remote Unix could potentially be used as an underlying run-time system for the `piostream` library but the current version of Remote Unix does not support inter-process communication.

In most existing systems, delivering input from the user console or a file[1] to threads executing on several different hosts is simply not possible. Instead the user must first read and buffer the input on the main host. They must then co-ordinate the sending and receiving hosts, and somehow transmit the data to the desired objects. Clearly this approach increases the complexity of the program and hence, makes the program harder to port and maintain. High maintenance levels impact the overall cost of a software project. Our implementation environment, ABC++, only provides `stdin` access to active objects that are specifically directed to execute on the main host. However, in a distributed system most active objects execute on remote hosts and hence they lack access to `stdin` on the main host.

[1] Through redirection to `stdin`.

Ideally, the user should be able to use `cin` remotely with a similar interface and behaviour as in a sequential program. The parallel input stream, `pistream`, is a component of the `piostream` library that is designed for easily performing input operations on remote active objects in a parallel or distributed environment. The `pistream` library reads data from `stdin` on the main host, and transmits the data to requesting active objects that may be executing on remote hosts. All buffering, synchronization, and the sending and receiving of data is handled transparently by the `pistream` library run-time system.

This chapter describes the design and implementation of the high-level object-oriented `pistream` construct. The chapter describes two different modes of data distribution between multiple active objects. Broadcast mode distributes identical data to requesting active objects while striped mode distributes the data sequentially to requesting active objects. A discussion is provided of the `pistream` client-server model and is followed by an analysis of various design issues. An overview of the `pistream` library prototype architecture and implementation issues is provided at the end of the chapter. One implementation problem found was that the IBM Cset1.1.1.4 for Solaris implementation of the `iostream` library cannot detect `EOF` properly for redirected empty files. This problem exists in the IBM Cset `iostream` library and subsequently exists in the `piostream` prototype because of the dependence of the `piostream` library on the `iostream` library.

4.1 Data Broadcasting and Striping

Input data is contained in a single stream from which various active objects request input. The input data from this single shared stream can be distributed in a variety of ways to active objects requesting input. Two common distribution techniques result from the two possible approaches to sharing the stream: each object can have an independent stream position or all objects can share the same stream position. If each object has an independent position in the stream, the resulting data distribution is called broadcasting. Broadcasting transmits each token of information to all requesting active objects. For example in a matrix multiplication program, all computing objects require a complete copy of one matrix. Conversely, if all active objects share a single stream position that progresses through the stream as data is extracted, the resulting data distribution is called striping. Striping transmits tokens of information to requesting active objects in a FCFS fashion. In the matrix multiplication example, a second matrix is usually split between several objects on a row by row, or a column by column basis. Striping allows each active object to input an entire row or column so that no other active object performing input receives the same row or column.

The broadcast and striped data distributions are illustrated with an example in

Figure 4.1. A string of input tokens is shown in Figure 4.1A. This string is read

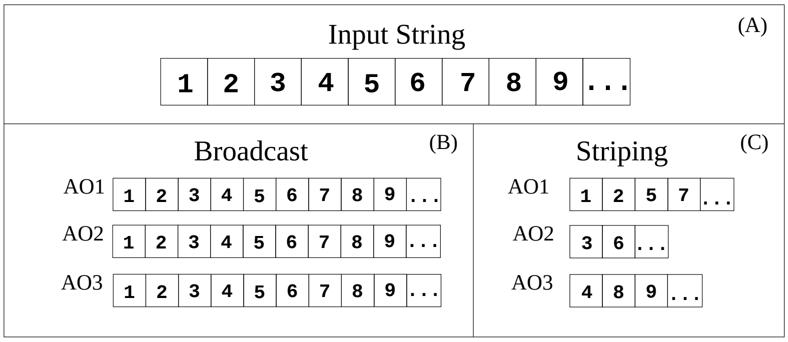

Figure 4.1: This diagram illustrates different approaches of distributing data between active objects. (A) represents a sample sequence of input tokens. (B) displays the data distribution between three active objects resulting from broadcasting. Broadcasting provides identical data to all requesting active objects. (C) displays an example data distribution between three active objects using striping. Striping provides sequential data to requesting active objects on a FCFS basis such that no active object receives the same data.

and distributed to three active objects that request input (as shown in Figure 4.1B and Figure 4.1C). When using a broadcast approach, as shown in Figure 4.1B, the input string is provided identically to all three active objects. Figure 4.1C displays an example of the data that might be received by the three active objects through the striped distribution technique. In this case the data is distributed and consumed on a FCFS basis with a shared stream position so that no active objects receive the same data.

The `pistream` design does not allow the use of both broadcast and striped data distributions in the same program execution. The semantic difference in the use of stream positions in each distribution mode presents a problem for run-time switching between data distribution modes. Switching from striped to broadcast mode is trivial because the single stream position used in striping would become the independent stream position for each active object in broadcast mode. Switching to striping mode from broadcast mode however, poses the problem of creating a single shared stream position from multiple independent stream positions. The value of the single stream position could be determined in one of several ways. For example, the stream position could be either the position of the object that requested the mode change or the position of the object that has read the most data. The use of the most advanced position would ensure that no active object's stream position regresses. Regressing in the stream could result in the same data being read a second time. The ideal stream position cannot be determined by the run-time system because only the user can decide the desired behaviour. Hence, the `pistream` design only allows one data distribution mode to be used in a single program execution. The data distribution mode is set with a command line parameter and defaults to the broadcast distribution mode.

Broadcast mode uses multiple independent stream positions for each active object performing input. An initial stream position must be set for each new active object. The easiest solution, and arguably the best solution, is to initialize each stream position to the beginning of the input stream. Therefore, each active object is ensured to receive identical data in broadcast mode. Assuming each active object makes identical input requests. However, this solution potentially requires the entire input stream to be buffered during program execution however, which can result in significant memory overhead. Even if all existing active object clients make progress reading data, a new client that requests data requires access to the entire sequence of input from the beginning token. A potential solution to the memory overhead required would be to cache the input buffer in a temporary file if a size limit is exceeded. The drawback to this solution would be the lower access speed of accessing the cached file. However, if the amount of input is sufficiently large, the user should use file I/O rather than redirection to `stdin`.

The `pistream` library support of broadcast and striped data distribution modes simplifies user programs. The `pistream` library transparently provides all necessary buffering and transmitting of data, in addition to providing the coordination between active objects required to distribute input data in either a broadcast or a striped fashion.

4.2 Pistream Architecture

As in the case of `postream`, the architecture of `pistream` is based on a client-server model. In this section we first present the client-server model and then provide a discussion of the issues involved in the design of the `pistream` construct. The `pistream` architecture uses a separate server object than the `postream` construct, due to the semantic differences in `pistream` and `postream` client-server interactions. As is the case with `istream`, the `pistream` object blocks when requesting data. Hence, the `pistream` RMIs are synchronous and conversely, the majority of the `postream` RMIs are asynchronous.

4.2.1 Client-Server Model

The active object server is instantiated on the main host so that the server can access `stdin` on the main host. The server is primarily responsible for two tasks. The server must read data from `stdin` and serve client RMI requests for input. The server must alternate between these two tasks by checking if any client RMIs are pending and by polling for data on `stdin`. The server continually reads data from `stdin`, stores it in the server's data buffer, and serves client RMI requests for input. If the server data buffer grows too large, it is cached to a temporary file on the main host. The input server has sole access to `stdin`. For this reason, the main program is also a client of the `pistream_server`. The directional flow of data in the client-server model is illustrated in Figure 4.2. As mentioned in the previous section, the client-server

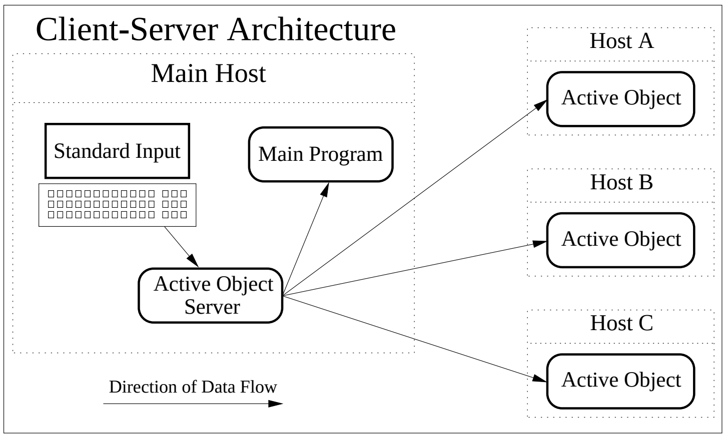

Figure 4.2: This diagram illustrates the directional data flow in the `pistream` client-server model. A dedicated active object server resides on the main host and reads input from `stdin` which is also on the main host. The server buffers and distributes data to requesting remote active objects and the main program.

model supports both the broadcast and striped distribution techniques because it has sole access to the input stream.

Another concern is if the server accepts a client RMI request for data that cannot be completed because no data is available for the requesting object. This occurs when the client's stream position is at the end of the server's data buffer (i.e., the client has read all of the input that has been made available to the server). In the sequential `iostream` library, the object requesting input blocks until data becomes available. Blocking until data is available in striped mode is an acceptable solution because the next input read must be given to the blocked client in order to comply with the

FCFS data distribution of striping. However, blocking is not acceptable in broadcast mode. Recall that broadcast mode uses multiple independent stream positions. Other client stream positions may not have reached the end of the server's data buffer, hence other client's RMI requests for data may still be served, even if the current client RMI request cannot be completed. The problem is therefore that the server can process only one RMI at a time (in order to satisfy FCFS) and while the currently executing client RMI cannot proceed (because no data is available), other client RMI requests can proceed because data may be available. This problem is solved through the use of nested RMI acceptance. Nested RMI acceptance allows an active object to accept and execute an RMI while an RMI request is being processed. For example, an active object could accept an RMI on the method `foo()` while method `foo()` could accept an RMI request on another method, including `foo()` itself.

Another concern is how the server should react if no RMIs are pending and no data is available on `stdin`. If the server continually loops while polling for data on `stdin` and for RMI requests in the message queue, busy waiting results. Busy waiting is the concept of keeping the CPU *busy* while it *waits* for some event. Busy waiting usually lowers performance and should be avoided if possible. Ideally, the server should block until data is available on `stdin` or a client RMI is requested. Blocking for both of these conditions simultaneously is not possible because one condition involves the run-time system (RMIs in the active object message queue) and the other condition involves an operating system device (data available on the `stdin` stream). Hence, the server must not block until data is available on `stdin` or until client RMI requests are received.

- If the server blocks until data is available on `stdin`, the server can become permanently blocked. If all the program data has been input when the server blocks, the server will never unblock. Subsequently, the blocked server cannot accept and serve client RMI requests. Therefore, the server must not block until data is available on `stdin`.

- It is acceptable for the server to block until a client RMI is received and input is being consumed in striped mode. The server blocks until an RMI is requested, which maintains the FCFS data distribution of striping. However, in broadcast mode, if the server blocks until an RMI is received, the server can enter a deadlocked state. Because broadcast mode allows nested RMI acceptance (to solve the no data available problem), it is possible that all active objects performing input could become blocked if their RMI requests become nested on the server. If no input data is available on `stdin`, the server will block until another RMI is requested by a client. However, it is possible, that no more RMIs will be requested. All the clients performing input could be nested already (and therefore blocked) and consequently no more RMIs will be requested. If the server blocks until another RMI is requested, it cannot input data from `stdin` that is required to complete the nested RMI client requests. The `pistream` system is now deadlocked because the server is waiting for a client RMI request and the client objects are blocked until the server reads more input data. Therefore having the server block on client RMI requests is also not a valid option for preventing busy waiting (in broadcast mode).

The solution used to prevent busy waiting is to temporarily block the server between polling attempts. The server provides a method to control the length of time the server blocks between polling attempts. If the user wants the input server to busy wait, they can set the server wait time to zero through the `pistream` client interface.

Another concern in the `pistream` library design is the amount of message passing between the client and server. Traditionally message passing is very expensive in a parallel and distributed system. Consider a line of input that is composed of several tokens (words, integers, and floats). Each `operator>>()` function invocation inputs a single token of data and therefore requires the client to communication with the server once for each token. In order to improve performance, the `pistream` object maintains a buffer on the remote host on which the client active object is executing when broadcast mode is used. The buffer caches an entire line of input. By buffering the line of input, the number of messages is reduced because the entire line is retrieved with one RMI request rather than using an RMI request to retrieve each individual token. Because each active object inputs the same data in broadcast mode, the ordering of input obtained by each object is not affected. However, caching is not supported in striped mode because the ordering of data would be affected. For example, consider two active objects that input integers from `stdin`. If caching is not used, the first object to input would read the first integer, the second object to input (not necessarily the second active object) would input the second integer. Caching of input would alter this because the first object would input the first integer but would also cache the first row of integers on the client side. The second object would subsequently read the first integer of the *second* row (and cache the remainder of the second row on the client side). Hence caching cannot be used in striped mode because it alters the order of input received.

However, because caching is not used in striped mode, `pistream` methods that query the state of the input stream, such as `eof()` and `peek()`, are not guaranteed to be valid for the querying object's next input operation. A second object may input from the shared stream and therefore change the state of the stream from the result sent to the first object.

The `pistream` client-server model is also extensible. The model supports the input of user-defined classes through operator overloading, the chaining of `operator>>()` functions, and the use of input manipulator functions. All necessary interactions between the client and server are performed transparently by the `piostream` library run-time system. The code at the user-level simply uses `pin` in the same fashion that `cin` is used in sequential programs. Unless explicitly stated, all client object actions stated here are performed by the run-time environment on behalf of the client objects.

4.2.2 Architecture Overview

The `pistream` client-server architecture has a client and a server component. The server runs on the main host. The client is a data member of each active object and resides where the active object is executing. The major components of the server and client are described in this section. The interface of the client and server can be found in Appendix A.

Server Architecture: The server component of the `pistream` library is primarily implemented within a single class, `pistream_server`, which uses several smaller classes for the underlying implementation. A single object of this class, `pi_server`, is instantiated to act as the input server. The `pistream_server` class contains a buffer for storing input for the program. This buffer grows dynamically when input is inserted in either the striped or broadcast distribution mode. In broadcast mode, a vector of stream positions is maintained, one per client. Each client's stream position is added to the vector when the run-time first requests an identification key on behalf of the client. An initial default size is used, and is doubled each time the vector becomes full. When a client requests data, a lookup is performed to determine the appropriate stream position. The server then seeks to the client's read position in the buffer. The corresponding data is then returned to the client. The client's new stream position is stored back into the appropriate vector location. In striped mode, a single stream position is used to read from the buffer. The server's stream position changes as each client's RMI request is served.

The implementation environment, ABC++, does not support nested RMI acceptance because the FCFS order of RMI completion cannot be guaranteed when using asynchronous RMIs. For example, consider an active object that requests RMIs asynchronously on another active object. If the accepting active object nests RMI requests and hence executes multiple RMI invocations simultaneously, the order of RMI completion will be LIFO (last-in first-out) instead of FCFS. However, if only synchronous RMIs are used, each active object can only synchronously request a single RMI at any one time (the object blocks until completion of the RMI) and therefore FCFS order can be maintained. The additional methods `PIO_accept()` and `PIO_accept_any()` were implemented to provide nested RMI acceptance. Nested acceptance is safe for use in the `pistream` library because the client-server model for the `pistream` construct uses only synchronous RMIs, hence FCFS order is maintained. Because these nested RMI acceptance methods are only used by the I/O library, the FCFS invocation behaviour of the active object model is not violated.

Polling for data on `stdin` is complicated due to the possibility of redirected files. Console input can be detected through polling the `stdin` read state using the UNIX system call `select()`. Unfortunately, this technique does not work correctly for redirected file input because the `select()` call incorrectly reads the availability of data on the `stdin` stream when a file is redirected. The `select()` call incorrectly returns the result that data is available, even if the `EOF` condition is present, and no more data is available. Conversely, the `iostream` method `eof()` can detect the `EOF` character, however, console input does not use an `EOF` value . Consequently, `eof()` cannot be used to determine data availability with console input. Hence, in order to detect available data from either the console or a redirected file, both approaches are combined. Data availability is detected through the read state of `stdin` and the state of the `EOF` bit.

The server also has a data member that defines the length of time the server should block between polling `stdin` and checking for client RMI requests. This time can be modified by the user using a public method. In the prototype implementation, temporary blocking is implemented using the UNIX system call `usleep()`. All public `pistream_server` methods are invoked through the `pistream` client interface by the underlying run-time system on behalf of the client. The `pistream_server` supports the following methods:

- *Request key* (`request_Key()`) — this public method allows a client to obtain a unique identification key for communicating with the server. This method is invoked by the run-time system the first time an input attempt is made by each active object. The method cannot be invoked when each active object is instantiated because the active object communication mechanisms may not have been instantiated at this time.

- *Request data* (`request_Data()`) — this public method is invoked by the client run-time system to request data. The client's index key and a delimiting character are provided as arguments. In broadcast mode, the server provides a string of data, delimited by the specified character — with the default being the new line character. The use of line oriented input supports a similar interface and behaviour as the `istream` library and is discussed further in following section. In striped mode the maximum length of the desired input string is also provided. This length allows the server to provide striped input by returning the properly sized string. The conversion of data from character string to C++ data type is handled on the client side of the `pistream` client-server model.

- *Extract delimiter* (`extract_Delimiter()`) — this private method is invoked by the server's `Read_data()` method. The method is used to extract the delimiting character from the server buffer. The delimiter is then concatenated with the `Read_data()` return value.

- *End of file* (`eof()`) — this public method is invoked by the run-time system and performs the same function as the `eof()` `istream` method. The method returns the condition of the `EOF` bit for the client's stream position in the server's data buffer.

- *Poll for data on* `stdin` (`poll_Available_Data()`) — this private method is invoked by the server to determine if data is available to be read from `stdin`. The technique used can detect input from either the console or a redirected file.

- *Read data* (`read_Data()`) — this private method is invoked by the server when data is detected on `stdin`. A line of data is read and inserted into the server's data buffer.

- *Set wait time* (`set_Wait_Time()`) — this public method is invoked by the run-time system to set the length of time the input server blocks between polling for data on `stdin` and handling RMIs in the message queue.

- *Block temporarily* (`wait()`) — this private method is invoked by the server if no RMIs are pending and no data is available on `stdin`. The method uses the `usleep()` system call to block for `wait_time` length of time. The `wait_time` value can be modified by the `Set_wait_time()` method.

- *Shut down server* (`terminate()`) — this public method is invoked by the run-time system to shut-down the server.

Client Architecture: The client component of the `pistream` library is implemented using one class, `pistream`. Extending the `istream` class through inheritance is not possible for the same reasons as discussed for `postream` in Section 3.3.4. Each client active object contains an object of the `pistream` class, named `pin`. This is implemented in a fashion which is similar to `postream`, by adding `pin` as a data member of the root class `Pabc` in ABC++.

The `pistream` class contains a buffer that is used to cache a line of input data. The `get()` and `getline()` methods allow the user to specify a delimiting character. This character is used to bound by context the data string extracted by the `get()` and `getline()` methods. The delimiter default value is the new line character `'\n'` which works well with the line-oriented caching of the `pistream` object in broadcast mode. However, if the user invokes `get()` or `getline()` with a different delimiter, the desired input string could require more data than is in the cache. This occurs if the cache size is smaller than the specified number of characters to be read and the delimiter is not encountered in the cache. For example, the requested data includes all the data in the current client cache *and* one or more characters that are still stored in the server's buffer. The run-time system transparently handles this problem by requesting additional data from the server until either the requested number of characters are read or the delimiting character is encountered. The requested data is concatenated by the run-time system with the initial cache data stored on the client to produce the proper result. This solution provides the same interface and behaviour as the standard `iostream` library `get()` and `getline()` methods. In striped mode, the maximum length of characters to be read is also specified to ensure the input is properly striped.

Other data members of the `pistream` class include an identification key provided by the `pistream_server` object and the handle of the `pistream_server` object. The `pistream` class supports both the input of user defined objects (through overloading), and manipulator functions. The interface and behaviour of the `pistream` methods are similar to their `istream` counterparts. The methods supported by the `pistream` class include the following:

- *Set address of* `pistream_server` (`set_PIS_Server()`) — this private method is invoked by the run-time system to set the `pistream_server` handle, which consists of the server process id (host name or Internet address) and memory address. This must be done by the run-time system for transparency reasons as discussed for the `postream` object in Section 3.3.4.

- *Input operator for predefined types* (`operator>> {predefined type}`) — this public method allows the client to extract data from the line of input cached by the client into the requested data type. In all input methods, if the input request cannot be completed because the client cache lacks enough data, the client run-time system transparently requests more data by invoking the `pistream_server`'s `request_Data()` method.

- *Input operator for manipulator functions* (`operator>> {manipulator function}`) — this public method is used in a similar manner as the `operator<<()` functions for manipulator functions described for the `postream` class. The passed manipulator function is invoked on the client side with the `pistream` object as an argument.

- *Input character(s)* (`get()`) — these public methods allow the user to input either a single character, or a number of characters until a delimiter character is encountered (Several overloaded versions exist. See [21](pp. 120) for more details). The delimiting character is not extracted from the input buffer similar to the `istream` `get()` method. Each `get()` method's interface and behaviour is similar to its respective `istream` counterpart.

- *Input next line* (`get_line()`) — these public methods allow the user to input a single line, or a number of characters up to the delimiting character. (Several overloaded versions exist. See [21](pp. 120) for more details). These methods have the same behaviour as the `pistream` `get()` methods except that the delimiting character is extracted and thrown away. Each `getline()` method's interface and behaviour is similar to its respective `istream` counterpart.

- *Examine next character* (`peek()`) — this public method allows the user to examine, without extracting, the next character of the input buffer. The returned value is of type integer in order to allow the `EOF` value to be returned.

- *Manipulator functions* (`ws`, `hex`, etc.) — these functions implement the `pistream` versions of the standard `istream` manipulator functions. These functions are invoked in a similar manner as their `istream` counterparts.

- *End of file* (`eof()`) — this public method allows the client to obtain the condition of the `EOF` bit for the client's stream position in the server's data buffer. The run-time system transparently invokes the server's `eof()` method.

- *Set wait time* (`set_Wait_Time()`) — this public method allows the client to set the length of time the server blocks between polling for data on `stdin` and handling RMIs in the server's message queue. The run-time system transparently invokes the server's `Set_wait_time()` method with the given value as an argument.

4.3 Summary

This chapter presents an overview of the design and implementation of the `pistream` component of the `piostream` library. The `pistream` library is a high-level object-oriented construct that simplifies user input operations and therefore, increases the portability and maintainability of the user program. Active objects that execute on remote hosts often cannot input using `stdin` because `cin` objects on the remote host are not associated with `stdin` on the main host. Users must provide complex buffering and coordination between active objects to facilitate data distribution to remote active objects. The `pistream` construct provides parallel and distributed input facilities for remotely executing active objects. The `pistream` run-time system transparently provides support for data transfer, buffering, and distribution in a broadcast or striped fashion. The `pistream` construct provides an interface and behaviour that is similar to the C++ `istream` construct. At the user-level, active objects simply use `pin` to input data from `stdin`. As in C++, `operator>>()` can be overloaded at the user-level. Manipulator functions are used as in C++. An example of the `pistream` syntax using the `pin` object is shown in Figure 4.3.

```
int mass;
float acceleration;
double velocity;
char buf[size];
•••
// Input mass, velocity and acceleration, skipping white space
pin >> ws >> mass >> velocity >> acceleration;
// Turn off white space skipping
pin.unsetf(ios::skipws);
// Set input server poll time to 0.5 seconds
pin.set_wait_time(500000);
// Read in data until end of file is reached
while (! pin.eof() ){
  pin.getline(buf, size);
  pout << "Input [" << buf << "] received." << endl;
}
```

Figure 4.3: This diagram illustrates the syntax used to obtain input using the `pistream pin` object. An integer, float, and double are input using a chained invocation. The chained `ws` manipulator is used to skip white space. White space skipping is turned off by directly invoking the `unsetf()` method and the `skipws` manipulator function. The `pistream` server poll time is set to 0.5 seconds (the wait time is in micro seconds). Finally the program reads in data in fixed size segments (because no white space is skipped) and printed to `stdout` using the `postream` object `pout`.

Chapter 5

Pfstream

Many parallel and distributed programs use file I/O for reading input and writing results. A major concern is that active objects executing on remote hosts may not be able to access the main host file system. The main host file system is considered the most important file system because it is the environment where the main program is executing. The NFS (Network File System) provides support for the sharing of different machine's file systems. It is possible however that file systems cannot be mounted for either security or administrative reasons. Another potential difficulty with the NFS is the coordination required to support active object file I/O with a shared single file pointer. The statelessness of the NFS server requires any coordination to be performed explicitly by the user. For these reasons the NFS is not a suitable solution for high-level distributed file I/O. A second concern is the low-level interface supported by many parallel and distributed systems. For example, PVM and MPI both require the user to explicitly convert data from C++ or user-defined types to character strings for output purposes. Hence in addition to the difficulty of parallelizing the program, the user must add extra code to perform any required data buffering, transferring, and synchronization between an object that has access to the main host file system and remotely executing active objects that perform I/O. This added complexity makes the user program more difficult to port and maintain.

The design of the proposed parallel file stream constructs alleviates the programmer's burden of buffering and transmission of data in parallel and distributed environments. As is the case in the sequential `iostream` library, three different high-level object-oriented file stream constructs are proposed: `pifstream` for input, `pofstream` for output, and `pfstream` for cases where both input and output operations are performed on the same file. These constructs comprise the `pfstream` library which is a high-level object-oriented solution that provides the familiar C++ `iostream` interface for file I/O. The proposed `pfstream` library allows multiple active objects to output to the same file, even if the active objects are not executing on hosts that have access to the main host's file system. The library also provides data origin identification for output written to files in the same manner as described for `postream`.

Many of the issues involved in the design of the `pfstream` library are similar to issues encountered in the `postream` and `pistream` constructs. This chapter describes the design of the object-oriented `pfstream` library with a focus on issues concerning the support of parallel and distributed file I/O. These issues include:

- The modes supported for multiple active objects to simultaneously access the same file. By simultaneous we mean that at least two client objects overlap the `open()` and `close()` file operations. The `pfstream` library supports independent and shared file stream positions.

- The tracking and representation of open files required to support shared access to the same file by multiple clients.

- The simultaneous use of different access modes on the same file by different active objects.

- The opening of the same file multiple times by the same active object.

An overview of the proposed `pfstream` client-server architecture and class hierarchy is provided at the end of the chapter.

5.1 Issues in the Design of the Pfstream Library

The architecture of the `pfstream` library is based on a client-server model. The single server may create a bottleneck that can impact the performance of the program. The objective of the `pfstream` library is not to provide a high-performance solution to the I/O bottleneck. Rather, the `pfstream` library is intended to support distributed I/O through high-level constructs, hence reducing the complexity of writing, debugging, and maintaining parallel and distributed programs. In this section we discuss the client-server model and the issues involved in the design of the `pfstream` library followed by an overview of the `pfstream` library architecture.

5.1.1 Client-Server Model

An active object server is instantiated on the main host so that the server can access the main host file systems. Although other hosts may have access to these file systems using for example NFS, only the main host is guaranteed to have access to its own filesystems. The server is responsible for performing I/O on files residing on the main host file system and serving client RMI requests. Unlike the `postream` and pistream constructs which must provide access to a well-defined number of I/O devices (stdout, stderr, and stdin), one pfstream object is instantiated for each file that is opened by an active object. The number of simultaneously open files is bound by the operating system's limit on the number of open file descriptors available to a single process (the active object server). Because the active object server handles all file I/O requests, the total number of open files for all clients is bound by this limit. The server is responsible for opening, reading, writing, and closing each file as requested by the client active objects. Although the main program can access the main host file system directly, it is also possible for the main program to act as a client of the server for access synchronization purposes and to provide a uniform interface for all client objects. The directional flow of data in the client-server model is illustrated in Figure 5.1. The server is responsible for accessing files for input,

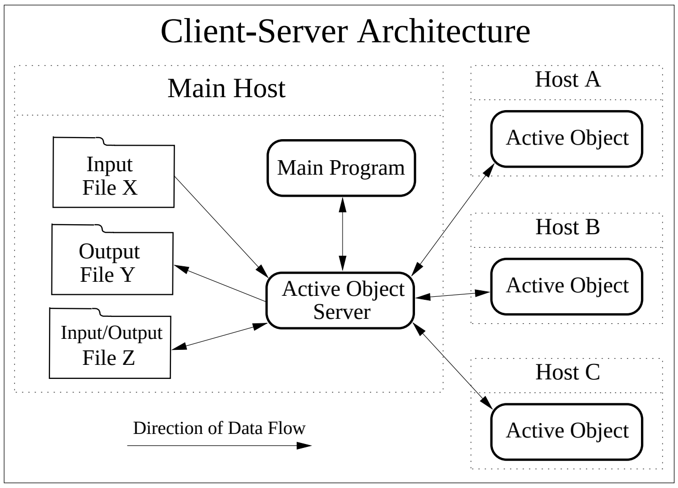

Figure 5.1: This diagram illustrates the directional data flow in the pfstream client-server model. A dedicated active object server resides on the main host and performs I/O on files on the main host file system. The server serves I/O requests from remote active objects and the main program. Unlike the postream and pistream server, the pfstream server is responsible for serving both input requests and output requests through instantiated pfstream objects (pifstream, pofstream, and pfstream).

output, and bi-directional I/O.

Access Modes: The pfstream library supports two access modes: independent access in which each client has an independent file stream position; and shared access in which all participating clients share the same stream position. An extra file open mode is provided that specifies the desired type of access. Independent and shared access are similar to broadcasting and striping which are described in Section 4.1. However, independent and shared access can also be applied to output and bi-directional file streams. Shared access provides synchronized interleaving of I/O requests for each client active object. The run-time system transparently provides the required active object coordination, buffering, and data transmission. Independent access supports independent stream positions for each client, but the user must ensure that the active objects do not interfere with each other which could result in the overwriting of data. The user must provide this synchronization because the run-time system cannot determine the behaviour desired by the user.

Tracking and Representation of Open Files: A major concern of the `pfstream` construct is determining when the same file is opened by different clients. Each file opened by the `pfstream` server must be tracked and identified. Several approaches are possible for representing files that are opened simultaneously by multiple clients. The first approach is to maintain a `iostream` `fstream` object for each client. A second approach is to use a single `fstream` object and a table of file stream positions for seeking within the file. Using multiple `fstream` objects is easier to implement than sharing an `fstream` object between clients (which may request different open modes). Moreover, the necessary seeking involved in the second approach could create significant overhead. However, using a separate `fstream` object for each client requires a larger number of file descriptors to be used, which could limit the `pfstream` library's usefulness because of the limitation on the number of files that can be opened simultaneously. A modification of the multiple `fstream` approach can eliminate this hindrance and is used for the `pfstream` design. This modified approach uses an `fstream` object to represent each client, but each `fstream` object shares the same file descriptor (assuming the same file is opened). Sharing file descriptors is accomplished with the `fstream` method `attach()` using a file descriptor of an opened file as an argument.

When an open file request is made by a client object, the server must search the table of file names to determine if the requested file is already open. If the file is not open, the server opens the file using an `fstream` object, stores the file's name, descriptor, and open modes. If shared access is specified, the server sets the number of file clients to one. If the file is already open, and the request is for independent access, a new `fstream` object is attached using the existing file descriptor. Clients using shared access to a file would use the same `fstream` object and subsequently the same file stream position. If the open request is for shared access, the server increments the number of clients using the file. Conversely when a client makes a close file request, the server deletes the `fstream` object (if access is independent) or decrements the appropriate file's client count (if access is shared). If the number of clients sharing a file reaches zero, the file is closed by the server and removed from the table of file names.

Conflicting Access Modes: The problem of conflicting open modes[1] is a more difficult issue. Two client objects may open a file with one or more conflicting open modes. For example, Client A may open file `foo` with the open modes: independent access and input only. Client B may subsequently (before Client A closes file `foo`) open the file with the open modes: shared access, output only, and append mode. In this case the open modes conflict (independent — shared, input only — output only, no append — append). The server has a variety of options including: using the file's original open modes (independent access and input only), changing the file's open modes to the requested open modes (shared access, output only, and append), permit each client to maintain a distinct set of modes that guide the client's access, or throwing a C++ run-time exception. Enforcing one client's open modes on a second client is not a valid solution because the second client will not have the expected semantic behaviour. Hence, the first two options are not suitable for our design. The use of client specific open modes simplifies the problem because only clients that share access can conflict. In shared access the conflict must be solved by the use of run-time exceptions. Even the use of shared and independent stream positions on the same file can be accommodated using client specific modes. All clients that request shared access must use the same open mode settings. The user is responsible in all cases for ensuring that the desired semantics are maintained by the client objects.

Multiple Open: Active objects can open the same file multiple times using one instantiation of a `pfstream` object for each file stream. Each `pfstream` object is treated as a separate client by the `pfstream` server.

5.1.2 Architecture Overview

The `pfstream` client-server architecture has a client and a server component. The server runs on the main host. Each client is instantiated by an active object to interact with a single file (at a time). The client resides where the active object is executing. The major components of the server and client are described in this section.

Server Architecture: The server component of the `pfstream` library is primarily a single class, `pfstream_server`. A single object of this class, `pf_server`, should be

[1]File open modes are used to specify the semantics desired for the file stream such as read only, write only, and open an existing file only. File open modes are described in Section 2.1.1

instantiated to act as the remote file I/O server. The server must maintain a table of open file names as well as a table of `fstream` objects and their corresponding open modes. The server provides public methods for opening, reading, writing, and closing files. The server also supports public methods for reading and setting the file stream position. These methods are invoked remotely by the client active object through the client `pfstream` object interface. The `pfstream_server` supports the following methods:

- *Open file* (`open()`) — this public method allows a client to open a file on the main host. An index value is returned to the client for subsequent communication with the server. If shared access is requested, and the file is already open, the client's file stream is attached to the corresponding file descriptor.
- *Attach file* (`attach()`) — this public method is invoked by the run-time system to attach the specified file descriptor to the client file stream.
- *Read data* (`read_Data()`) — this public method is invoked by the run-time system to read and return data from the specified file
- *Write data* (`write_Data()`) — this public method is invoked by the run-time system to transfer data to the server for output to the specified file.
- *Read stream position* (`tell()`) — this public method is invoked by the run-time system to read the stream position in the specified file. Either the get or put stream position is specified by a `boolean` argument.
- *Set stream position* (`seek()`) — this public method is invoked by the run-time system to change the stream position in the specified file. Either the get or put stream position is specified by a `boolean` argument.
- *End of file* (`eof()`) — this public method is invoked by the run-time system and returns the status of the `EOF` bit for the specified file.
- *Close* (`close()`) — this public method is invoked by the run-time system to close the specified file. The server closes the specified file if the requesting client is the last file stream attached to the file.
- *Shut down server* (`terminate()`) — this public method is invoked by the run-time system to shut down the server.

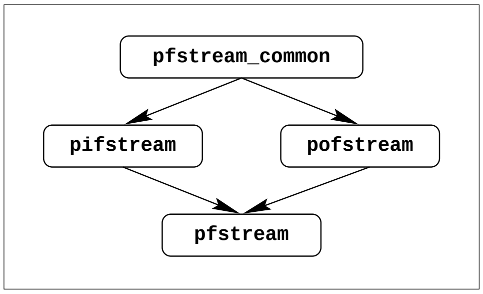

Figure 5.2: This diagram illustrates the `pfstream` class hierarchy. The `pfstream_common` base class defines the required communication and synchronization facilities required to support distributed file I/O. The derived classes `pifstream` and `pofstream` define the input and output methods required to support the `iostream` file I/O interface. The `pfstream` bi-directional class uses multiple-inheritance to derive both the `pifstream` and `pofstream` class functionality.

Client Architecture: The client component of the `pfstream` library consists of four classes: `pfstream_common`, `pifstream`, `pofstream`, and `pfstream` which are illustrated in Figure 5.2. The `pfstream` class hierarchy is similar to the C++ `iostream` `fstream` class hierarchy. The `pfstream_common` class is the base class of the hierarchy and encapsulates the communication and coordination facilities required to support distributed file I/O. The input file stream, `pifstream`, and the output file stream, `pofstream`, are both derived from the `pfstream_common` class. The `pifstream` and `pofstream` classes define the methods that support the `iostream` file I/O interfaces. The bi-directional file I/O class `pfstream` derives both the `pifstream` and `pofstream` interfaces through multiple inheritance. Extending the C++ `fstream` classes through inheritance to support distributed file I/O is not possible because of the problems caused by the non-member function `operator<<()` and `operator>>()` interface as discussed in Section 3.3.4 for the `postream` class.

The data members of the `pfstream_common` class include a file index and the handle of the `pfstream_server` object. The file index is used to identify the appropriate file in the `pfstream_server`'s table of file names. The interface and behaviour of all the `pfstream` class (`pofstream`, `pifstream`, and `pfstream`) methods are similar to their `fstream` counterparts. The `pfstream_common` class supports:

- *Set address of* `pf_server` (`set_PFS_Server()`) — this private method is invoked by the run-time system to set the `pfstream_server` handle which consists of the server run-time host and memory address.

- *Open file* (`open()`) — this public method allows the client to open a file on the main host. The file is specified by either file name or a previously opened file's descriptor. The run-time system transparently invokes the server's `open()` method.

- *Close file* (`close()`) — this public method allows the client to close the specified file. The run-time system transparently invokes the server's `close()` method.

- *Attach file stream* (`attach()`) — this public method allows the client to attach the file stream object to the specified file descriptor corresponding to an opened file on the main host. The run-time system transparently invokes the server's `attach()` method.

- *End of file* (`eof()`) — this public method allows the client to obtain the condition of the `EOF` bit for the client's stream position in the specified file. The run-time system transparently invokes the server's `eof()` method.

- *Read Stream Position* (`tell()`) — this public method allows the client to read the position of the client's get or put pointer in the specified file. The run-time system transparently invokes the server's `tell()` method.

- *Set Stream Position* (`seek()`) — this public method allows the client to change the position of the client's get or put pointer in the specified file. The run-time system transparently invokes the server's `seek()` method.

The `pofstream` class has the same interface as the `pfstream_common` base class and the `postream` class. Similarly, the `pifstream` class has the same interface as the `pfstream_common` base class and the `pistream` class. The `pfstream` class inherits the interface of both the `pofstream` and `pifstream` base classes.

5.2 Summary

This chapter presented an overview of the design of the `pfstream` component of the `piostream` library. The proposed `pfstream` constructs provide parallel and distributed I/O facilities for remotely executing active objects that access files with shared or independent file stream positions. The `pfstream` library supports a similar interface and behaviour as the C++ `fstream` library. At the user-level, active objects simply instantiate a `pfstream` object (`pifstream`, `pofstream`, or `pfstream`) and input or output data in a similar manner as in C++. I/O for arbitrary objects is supported through overloading of `operator<<()` and `operator>>()` functions. Manipulator functions can be used as in C++. The `pfstream` run-time system transparently provides support for data transfer, buffering, and synchronized file output operations.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis we describe the design and prototype implementation of an object-oriented C++ streams library designed for use in parallel and distributed environments. The `piostream` library provides support for performing I/O operations with active objects that may be executing remotely. Without the `piostream` library, the user is often required to buffer data and coordinate each active object before sending and receiving data from the I/O devices to the remote objects performing I/O. Moreover, the user must explicitly direct the active objects to send and receive the data. This explicit buffering, synchronization, and communication increases the program's complexity. In turn the increased complexity lowers the program's portability and maintainability which makes writing, testing, and debugging the program more difficult. A major portion of the cost of the software development life cycle can be attributed to the maintenance of a software program after its implementation [14, 23]. Hence, significant development costs can be reduced by increasing the maintainability of a software program.

The `piostream` library provides support for active objects that execute remotely to perform I/O operations that use `stdin`, `stdout`, `stderr`, and the file-system on the main host as if these devices were local to the executing objects. High-level constructs are provided for output (`pout`, `perr`), input (`pin`), and file I/O (`pfstream`, `pofstream`, `pifstream`). At the user-level, these constructs are used in the same manner as their `iostream` counterparts (`cout`, `cerr`, `cin`, and `fstream`). The `piostream` library transparently provides any buffering, synchronization, and data transmission required for active objects that may execute remotely to perform I/O operations using the main host I/O devices.

The aspects that make the `piostream` library particularly useful for debugging, executing, and maintaining parallel and distributed programs and hence the main contributions of this thesis are:

- The `piostream` library provides high-level constructs `pin`, `pout`, `perr`, and `pfstream` so that users are not required to construct, send and receive elaborate messages in order to perform I/O with active objects that may be executing on remote hosts. The `piostream` library transparently performs all necessary buffering, synchronization and data transmission. Therefore, user programs are reduced in complexity which results in increased portability and maintainability.

- A simple and familiar interface is provided to reduce the potential for errors that would be caused by learning and using a new interface for I/O operations. The `piostream` library supports the same interface as the standard C++ `iostream` library including the chaining of `operator<<()` and `operator>>()` functions, support for user-defined classes through overloading, and all `iostream` manipulator functions.

- Unlike many existing systems, the `piostream` constructs are built using and are fully compatible with standard C++ which means no special compiler support or preprocessing is required.

- Synchronization is an integral part of the `piostream` library and the underlying run-time system. Therefore, the user is not required to synchronize access between multiple objects and I/O streams on the main host.

- Mechanisms are provided for automatically and transparently identifying the source of active object output. Output being printed by active objects executing on multiple remote hosts is collected, collated and printed to a screen or file on the main host along with information identifying the host name, process id, and thread id of the active object performing the output.

- Input and output constructs can be used on any remote host. The `piostream` library provides mechanisms for distributing input in either a broadcast or striped fashion. The `piostream` library transparently provides remote active objects with access to `stdin` on the main host. This access is simply not possible in most existing systems with users being required to explicitly read, buffer and send the data to each remote active object.

- Access to the main host file system is provided for active objects executing on remote hosts. Cross-mounting different file systems may not be possible for administrative and security issues. The `piostream` library transparently supports synchronized access to files. Executing active objects on remote hosts provides the necessary framework for the piostream library to support file I/O on the main host.

The piostream library demonstrates that a familiar and natural interface can be provided for remote I/O operations. The design of the piostream library is object-oriented and encapsulates the buffering, synchronization and communication required to support remote I/O operations. Hence, the piostream library is portable and can be used to support remote I/O operations for any parallel and distributed system using the active object model of concurrency. Although the design and implementation of the piostream library is described in terms of C++ and the active object model of concurrency, the library can be adapted for any object-oriented concurrency model. Moreover, the piostream library demonstrates that remote I/O can be supported without requiring special compiler support or language extensions. The piostream library is implemented in and is fully compatible with standard C++.

The design process of the piostream library illustrates the complexity of providing useful remote I/O facilities such as output data interleaving and input data distribution modes. The required level of control over the flow of data to support these facilities introduces bottlenecks through the client and server interactions. The performance impact of these bottlenecks is a concern during the design of the piostream library. Subsequently the use of shared locks and synchronous RMIs is limited in the piostream design because of their impact on the parallelism of the program.

The design of high-level file I/O constructs illustrates the difficulty involved in supporting bi-directional file I/O operations. Many of the synchronization issues involved in bi-directional file I/O depend on user semantics. For example, the synchronization of file data between active objects that read and write to the same file is largely the user's responsibility. Enforcing an explicit synchronization scheme will not provide the desired semantics for all users. Subsequently, the piostream library strives to provide an object-oriented flexible solution to this and other problems encountered in supporting file I/O in a parallel and distributed environment.

6.2 Future Work

This thesis presents the design of the piostream library. A prototype of the piostream model was implemented within the ABC++ concurrent library to demonstrate that the piostream library design is sound and that it can be applied to a parallel and distributed system based on the active object model. Many of the existing iostream methods have been implemented to provide a working prototype. The next logical progression for the piostream library involves a full implementation that would include:

- Implementing all existing `iostream` methods and providing full `iostream` functionality within the `piostream` architecture. Because of the size of the `iostream` library, the full `iostream` interface is not supported in the prototype. Instead partial support was implemented to demonstrate the `piostream` library design's feasibility.

- Developing methods that provide the user with some control over buffer sizes. For example, the method `pistream::set_wait_time()` provides the user with control over the length of time between `pistream_server` polling attempts. Similar methods should be developed that provide control over the size of the `pistream`, `postream`, and `pfstream` client and server buffers.

- Examine the feasibility of integrating the different parallel server objects into a single entity. Initial study of this issue suggests that the parallel input, output and file I/O client-server semantics are not compatible. The input server RMIs are synchronous (the client active objects block while waiting for input data) while the output server RMIs are asynchronous (the client active objects block only long enough to send the output data to the server). The use of RMI polling and nested RMI acceptance however, could provide the necessary tools to design a single parallel I/O server. The use of a single server for I/O however, increases the potential bottleneck in the distribution of data between remote active objects and the main host.

- Implement run-time exception handling for the `piostream` library. Most exceptions can not be recovered from and therefore output an error message to `stderr` before shutting down the `piostream` run-time system. The prototype supports some exceptions which are thrown by the server if an invalid client index is received.

- Conduct performance tests comparing equivalent user programs that use the `piostream` library and the `iostream` library. We wish to show that the `piostream` run-time support of the necessary data buffering, transmission, and synchronization is comparable in performance with user-level solutions using only the `iostream` library constructs. The testing should compare the performance of a program using the `piostream` library with a program that uses the `iostream` constructs along with user-level buffering, synchronization, and the explicit sending and receiving of data between active objects.

The `piostream` library design allows tasks that may be executing remotely to access `stdout`, `stderr`, `stdin`, and the file system on the main host for I/O purposes. Data buffering, transmission, and object synchronization is transparently provided by the piostream run-time system. Safe, multi-threaded access to stdin, stdout, stderr and the filesystem on the main host is supported using an interface and behaviour similar to the standard C++ iostream library. At the user-level, the user performs I/O with pin, pout, perr, and pfstream in the same manner as their iostream counterparts.

The piostream library demonstrates that parallel and distributed I/O can be supported using a standard and familiar interface. Moreover, parallel and distributed I/O can be supported without the use of compiler and language extensions. The piostream library provides intuitive and powerful high-level object-oriented constructs that can offer significant benefits to programmers when writing, debugging, executing and maintaining parallel and distributed programs.

Bibliography

[1] E. Arjomandi, W. O'Farrell, I. Kalas, G. Koblents, F.C. Eigler, and G. Gao. ABC++: Concurrency and Inheritance in C++. *IBM Systems Journal*, 34(1):120–136, 1995.

[2] E. Arjomandi, W. O'Farrell, and G.V. Wilson. Smart Messages: An Object-Oriented Communication Mechanism. In *2nd Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 233–240, Toronto, Canada, June 1996.

[3] D.T. Barnard, R.C. Holt, and J.N.P. Hume. *Data Structures: An Object-Oriented Approach.* Holt Software Associates, Toronto, 1995.

[4] B. Eckel. *Thinking in C++.* Prentice Hall, New Jersey, 1995.

[5] M.A. Ellis and B. Strousstrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[6] MPI Forum. MPI-2: Extensions to the Message-Passing Interface. Technical report, (available at http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html), 1997.

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing.* The MIT Press, Cambridge Massachusetts, 1994.

[8] G. A. Geist and V. S. Sunderam. Network-based concurrent computing on the PVM system. *Concurrency: Practice and Experience*, 4(4):293–311, June 1992.

[9] J. Gotwals, S. Srinivas, and D. Gannon. pC++/streams: a library for I/O on complex distributed data structures. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 11–19, Santa Barbara, July 1995.

[10] J. Gotwals, S. Srinivas, and S. Yang. Parallel I/O from the User's Perspective. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 129–137, 1995.

[11] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of USENIX*, 1994.

[12] M. Litzkow. Remote Unix - Turning Idle Workstations into Cycle Servers. In *Proceedings of Usenix Summer Conference*, pages 381–384, June 1987.

[13] M. Litzkow, M. Livny, and M. W. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.

[14] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, New Jersey, 1988.

[15] N. Nieuwejaar and D. Kotz. Low-level interfaces for high-level parallel I/O. In *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 47–62, April 1995.

[16] W.G. O'Farrell, F.Ch. Eigler, I. Kalas, and G.V. Wilson. *An Introduction to the IBM Parallel Class Library for C++*. ABC++ Version 1, Release 1, IBM Canada, 1995.

[17] R. Sandberg. The Design and Implementation of the Sun Network File System. In *USENIX Association Conference Proceedings*, pages 119–130, January 1985.

[18] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, Cambridge Massachusetts, 1996.

[19] Sun Microsystems, Inc. C++ 4.1 library reference manual. pages 81–99, Mountain View, California, November 1995.

[20] Sun Microsystems, Inc. The NFS Distributed File Service. Technical report, (available at http://www.sun.com/software/white-papers/wp-nfs), March 1995.

[21] S. Teale. *C++ IOStreams Handbook*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1993.

[22] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992. ACM Press.

[23] K. Walden and J. Nerson. *Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems*. Prentice Hall International (UK), 1995.

Appendix A

Piostream Library Interface

The following appendices provide the `piostream` library interface supported in the prototype implementation. The complete `piostream` library code cannot be provided here for proprietary reasons pertaining to the ABC++ concurrency library.

A.1 Postream Class Interface

```
// Constructor
postream(output_destination);

// Destructor
~postream();

// Create pointer to Postream_Server
void set_POS_Server(virtual_processor_id, void *);

// Set info of active object on Postream_Server
void set_Identification();

// Send data to server
void send_Data();

// Send flag to server
void send_Flag(ostream_flag);

// setf and unsetf
void setf(long);
```

```
  void unsetf(long);

  // width, fill and precision
  void width(int);
  void fill(char);
  void precision(int);

  // Friend functions
  friend postream & operator<<(postream&, const int &);
  friend postream & operator<<(postream&, const float &);
  friend postream & operator<<(postream&, const double &);
  friend postream & operator<<(postream&, const char &);
  friend postream & operator<<(postream&, const ostream &);
  friend postream & operator<<(postream&, ostrstream &);
  friend postream & operator<<(postream&, const char *);
  friend postream & operator<<(postream&, postream & (*f)(postream &));

// Manipulator functions
  friend postream & endl(postream &);
  friend postream & ends(postream &);
  friend postream & flush(postream &);
  friend postream & hideid(postream &);
  friend postream & showid(postream &);
  friend postream & showbase(postream &);
  friend postream & noshowbase(postream &);
  friend postream & showpos(postream &);
  friend postream & noshowpos(postream &);
  friend postream & uppercase(postream &);
  friend postream & nouppercase(postream &);
  friend postream & showpoint(postream &);
  friend postream & noshowpoint(postream &);
  friend postream & skipws(postream &);
  friend postream & noskipws(postream &);
  friend postream & left(postream &);
  friend postream & right(postream &);
  friend postream & internal(postream &);
  friend postream & scientific(postream &);
  friend postream & fixed(postream &);
  friend postream & dec(postream &);
  friend postream & oct(postream &);
```

```
  friend postream & hex(postream &);

// Manipulators with parameters
  friend postream & setfill(postream &, char &);
  friend postream & setprecision(postream &, int &);
  friend postream & setw(postream &, int &);
```

A.2 Postream Server Class Interface

```
  // Constructor
  Postream_Server();

  // Destructor
  ~Postream_Server();

  // Main function
  int main();

  // Accept data from the client object
  // Index, Data packet, Sequence Complete?
  void transfer_Data(int, Container, boolean);

  // Index, Enumerated Flag
  // Accept manipulator enumerated flag from the client object
  void transfer_Flag(int, ostream_flag);

  //  Process id
  //  Thread id
  //  Host name
  //  Standard out or Standard Error
  int add_New_Active_Object(int, int, Container, output_destination)

  // Function used to flush all buffers in case of server shutdown
   void flush_All();

  // This function is used to shut down the server by the run-time system.
  void terminate();
```

A.3 Pistream Class Interface

```
// Constructor
pistream(boolean);

// Destructor
~pistream();

// Create pointer to Pistream_Server
void set_PIS_Server(virtual_processor_id, void *, pistream_mode);

// Get index from pistream_server
void get_Indentification();

// Get functions
int get();
void get(char* ptr, int count, char delim='\n');
void get(streambuf& sbuf, char delim='\n');
void get(char& ch);
void get(unsigned char& ch);
int peek();

// Getline functions
void getline(char *ptr, int count, char delim = '\n');
void get_line(int size, char delim = '\n');

// EOF function for checking server status
boolean eof();

// Set wait time for server polls
void set_wait_time(float);

// Friend functions
friend pistream & operator>>(pistream&, const float &);

friend pistream & operator>>(pistream&, const double &);
friend pistream & operator>>(pistream&, const char &);
friend pistream & operator>>(pistream&, const istream &);
friend pistream & operator>>(pistream&, const istrstream &);
friend pistream & operator>>(pistream&, pistream & (*f)(pistream &));
```

A.4 Pistream Server Class Interface

```
// Constructor
Pistream_Server();

// Constructor
Pistream_Server(pistream_mode);

// Destructor
~Pistream_Server();

// Poll for data on stdin
int poll_Data_Available();

// Block for data on stdin temporarily
void wait_Data_Available();

// Read data from stdin
void read_Data();

// Main function
int main();

// Get delimited sequence of characters
// Index, Size of buffer, Delimiter
Container get_Line(int, int, char);

// Provides a method of determining under broadcast mode whether
// more data exists for the server to extract
boolean eof(int id);

// Provides a method of setting how long the server should sleep
// between polling stdin and checking for client requests
// Time is in seconds
void set_Wait_Time(float);

// Get key from server
```

```
int request_Key();

// This function will check if the delimiter was handled properly
// Buffer, Size of buffer, Delimiter
void check_Delimiter(char, int, char);

// Function used to increase size of storage buffer
void increase_Buffer_Size();

// This function is used to shutdown the server by the run-time system
void terminate();
```