Flick: The Flexible IDL Compiler Kit
Version 2.1

User's Manual

Flux Research Group
Department of Computer Science
University of Utah

`http://www.cs.utah.edu/flux/`
`flick@cs.utah.edu`

November 1999

Contents

Preface

Flick is a flexible and optimizing compiler for interface definition languages (IDLs). Like a traditional IDL compiler, Flick reads a high-level interface specification and from that produces C or C++ language *stubs* to implement communication between client and server programs or between distributed objects. Unlike traditional IDL compilers, however, Flick generates speed-optimized code for a wide variety of IDLs, stub styles, and communication infrastructures. Flick can generate optimized code for many different applications such as ONC RPC (Sun RPC) systems, third-party CORBA ORBs, and other, more specialized situations. In sum, Flick is a "kit": the user picks the IDL, language mapping, and transport components that are required in order to implement the stubs for the user's particular application.

This manual describes how to build Flick, how to run the various compiler passes, and how to use the generated stubs. A simple client/server phonebook is presented and implemented in three different ways: as an ONC RPC (Sun RPC) application, as a CORBA C application, and as a CORBA C++ application. The examples are presented in detail and illustrate how users can make use of Flick-generated stubs in their own programs.

Chapter 1

Introduction

This guide assumes that the reader is familiar with the basics of distributed application programming: RPCs, RMIs, IDLs, and the various distributed computing standards such as CORBA and ONC RPC. Readers who are unfamiliar with distributed computing should review these terms before proceeding through this manual.

1.1 What is Flick?

Flick is a flexible and optimizing compiler for interface definition languages (IDLs). The name "Flick" is an acronym for "Flexible IDL Compiler Kit." Like a traditional IDL compiler, Flick can read a high-level specification describing an interface to a software component or module, and from that, produce special functions called *stubs* to implement the interface in the C or C++ programming language. The stubs implement remote procedure call (RPC) or remote method invocation (RMI) semantics and allow separate client and server processes to communicate through what appear to be ordinary, local procedure calls or method invocations.

Unlike traditional IDL compilers, however, Flick is implemented as a set of programs, each implementing a single phase of compilation. These separate programs allow Flick to support features beyond the capabilities of ordinary IDL compilers:

- *Multiple IDLs.* Most RPC and distributed object systems, including CORBA and ONC RPC (Sun RPC), specify *an* IDL and from then on refer to it as *the* IDL. Flick, however, does not introduce a new IDL but instead supports a number of existing ones side-by-side. Any IDL supported by Flick can be used to generate stubs for any environment (with some limitations). Supporting multiple IDLs allows for backward compatibility and migration of existing programs, and also allows developers to choose the best IDL for the task at hand.

- *Sophisticated code generation and optimization.* Flick was originally targeted toward communication in microkernel-based systems. In that environment, Flick provides the user-level glue between the microkernel's primitive IPC facilities and user-level application code. This target environment implies that Flick-generated stubs must be *very fast*: good microkernels always have extremely fast and tightly optimized primitive IPC paths, so inefficient stubs could easily raise the effective IPC time by an order of magnitude. Today, Flick produces optimized code for *all* of its supported transport facilities including TCP-based transports and lower-level, microkernel-based IPC.

- *Multiple back ends.* Finally, while many RPC and distributed object systems claim to support multiple transport protocols, the protocols they support are generally only slight variations of each other. For example, ONC RPC supports multiple transports in the sense that it can pass messages across either

TCP or UDP links. However, the basic data encoding scheme used (XDR) is the same in each case, as are the stubs generated by the IDL compiler. The multiple protocol support is really just a stream-based abstraction layer in the RPC runtime support library. There is little hope, for example, that the ONC RPC system could easily be made to support the CORBA Internet Inter-ORB Protocol (IIOP) even though both are based on TCP.

Flick, on the other hand, aggressively supports multiple protocols by allowing different stub code generators (back ends) to be used interchangeably. Each back end has complete control over the RPC protocols that it implements: the back end determines not only the underlying transport protocol (e.g., TCP, UDP, Mach 3 messages, direct function call, etc.) but also the data encoding scheme used (e.g., Sun's XDR, DCE's NDR, CORBA's CDR, "dumb" architecture-specific encoding, etc.) and the high-level RPC protocol layered above the low-level stream or message protocol (e.g., ONC RPC, CORBA IIOP, Mach 3's MIG protocol, etc.). Flick contains several back ends supporting extremely diverse protocol combinations.

Flick is being developed as part of the Flux Operating Systems Project at the University of Utah. However, Flick is intended to be an open "kit" that others will extend and adapt for their own projects.

1.2 Comparisons With Other Object, RPC, and RMI Systems

1.2.1 CORBA

Flick is not a drop-in replacement for a complete CORBA ORB. Rather, Flick is an IDL compiler — *one part* of a complete ORB — that generates code that works in conjunction with an ORB runtime library. Flick comes with a lightweight ORB runtime for C language stubs (described in Section 4.2), and this runtime provides only what is required for basic CORBA communication. Most importantly, Flick's C language runtime does not provide any "CORBA services," nor does it implement the Dynamic Invocation Interface (DII) or the Dynamic Skeleton Interface (DSI), etc.

Although Flick itself comes with minimal runtime support, Flick can be used in conjunction with more complete third-party ORB runtime environments. For instance, Flick's C++ components are designed to generate optimized stubs that work with TAO, the real-time ORB from Washington University in St. Louis.[1] In this case, you get the best of both worlds: Flick-optimized stubs and a complete CORBA development environment.

Flick provides interoperability, speed, and flexibility. Clients and servers that use Flick-generated stubs can communicate with other CORBA-based clients and servers through the Internet Inter-ORB Protocol (IIOP). Moreover, because Flick-generated code is optimized, clients and servers that use Flick-generated stubs can run faster than their counterparts that use stubs generated for other ORBs. Finally, Flick's kit architecture makes Flick suitable for a wide variety of environments. Flick can be used to generate code for TAO, or for its own minimal CORBA runtime, or for Mach, or for ONC RPC — in short, for any environment for which there exists Flick compiler components.

1.2.2 ONC RPC (a.k.a. Sun RPC, `rpcgen`)

Just as Flick is not a drop-in replacement for a complete CORBA ORB, neither is Flick a drop-in replacement for Sun's `rpcgen` program, the widely available compiler for the ONC RPC IDL. Flick-generated stubs can provide the same interfaces as those produced by `rpcgen`; further, those stubs can produce the same

[1]TAO is available from `http://www.cs.wustl.edu/~schmidt/TAO.html` and `ftp://ace.cs.wustl.edu/pub/ACE/`. Flick generates stubs that work with TAO version 1.0.

on-the-wire messages. But inside, Flick-generated stubs are very different than their `rpcgen`-generated counterparts. Flick does not, for example, provide an interface to XDR streams like those used by `rpcgen`-generated stubs. Flick does not provide the full ONC/TCP runtime. Rather, Flick implements the minimum "stuff" necessary for the top-level interface to ONC-style RPC.

So why use Flick instead of `rpcgen`? Performance! Flick-generated stubs are interoperable with `rpcgen`-generated code, but can encode and decode messages up to 17 times more quickly. ONC RPC clients and servers that make use of the high-level RPC interface, and not the lower-level interfaces offered by the ONC specification, can generally be recompiled or relinked without modification to use Flick-generated stubs, and thereby achieve greater network throughput.

1.2.3 Mach and MIG

Mach is a microkernel-based operating system that makes extensive use of IPC. The IDL compiler that creates client/server communication code for Mach is called MIG, the Mach Interface Generator. MIG is tightly integrated with its target language (C) and the Mach operating system. Flick supports nearly all of the presentation and transport features that MIG offers for Mach-based communication. In addition, Flick supports other IDLs to define Mach client/server communication over Mach ports, and this means that Flick supports notions that MIG does not: for instance, complex data types!

Chapter 2

Compiling and Installing Flick

This chapter is adapted from the `INSTALL` *file located in the root of the Flick distribution source tree and from the* `README` *file located in the* `test` *directory.*

Flick 2.1 has been successfully built and tested on the following platforms:

Operating System	Host/Processor	Compiler
FreeBSD 3.0-CURRENT	Intel Pentium II	gcc/g++ 2.95.1
FreeBSD 3.0-CURRENT	Intel Pentium II	gcc/g++ 2.7.2.1
Linux 2.2.12	Intel Pentium II	egcs 1.1.2
SunOS 5.7 (Solaris)	Sun 4u SPARC	gcc/g++ 2.8.0
SunOS 5.7 (Solaris)	Sun 4u SPARC	Sun WorkShop C/C++ 5.0
SunOS 5.7 (Solaris)	Sun 4u SPARC	Sun WorkShop C/C++ 4.2
4.3+ BSD	HP 9000/J210XC PA-RISC 1.1	gcc/g++ 2.7.2
IRIX64 6.5	SGI Origin 2000	MIPSpro 64-bit C/C++ 7.30
Windows 95	Intel Pentium II	egcs 1.1 with Cygnus' Cygwin library b20.1
Windows NT 4.0	Intel Pentium Pro	egcs 1.1 with Cygnus' Cygwin library b20.1

FreeBSD is the current Flick development platform. Flick-generated stubs have also been tested on Mach/Lites and our Fluke OS. If you build Flick on a platform other than the ones listed above, the Flick developers would love to know about it. Of course, if the build fails, they would like to know about that, too. Send build reports, bug reports, praise and damnation to `flick-bugs@cs.utah.edu`.

2.1 Building the Flick Programs

In order to build Flick you will need:

An ANSI C compiler — `gcc` 2.7.2.1 is our usual compiler. Flick has previously been built with `gcc` 2.6.3 and 2.5.8.[1]

[1] Using `gcc/g++` 2.5.8 may cause problems with debugging. Flick's data structures are not fully specified in the debugging information output by early versions of `g++`. This causes `gdb` not to examine many data structures fully. We recommend that you use `g++` 2.7.2.1 in order to avoid this problem.

An ANSI C++ compiler	`g++` 2.7.2.1 is our usual compiler. Again, Flick has worked with `g++` 2.6.3 / 2.5.8 in the past.
GNU `make`	Versions 3.69, 3.74, 3.75, and 3.77 have been tested.
`flex` or `lex`	`flex` versions 2.4.6, 2.5.2, and 2.5.4 have been tested. Vendor-provided versions of `lex` should work, but this has not been extensively tested.
`bison` or `yacc`	`bison` version 1.25 has been tested; version 1.22 is known not to work. Vendor-provided versions of `yacc` should work, but have not been extensively tested.
Disk space	You will need ~140MB of free disk space for a fully debuggable Flick object tree, and an additional 100–200MB in order to run the Flick tests.

Once you have those tools, you must unpack the Flick distribution `tar` file and create a build tree to contain the object files. If you create a build directory called `obj`, your directory setup would look as follows:

`~/flick-2.1`	The Flick source tree, unpacked from `flick-2.1.tar.gz`.
`~/obj`	The Flick object tree, which you will use to build Flick.

In that configuration, you would type the following commands in order to first configure Flick for your operating system and then compile all of the Flick tools:[2]

```
cd ~/obj
../flick-2.1/configure
make
```

If all goes well, links to the suite of Flick programs will be installed in `~/obj/bin`. Section 2.3 below explains how to install the Flick tools in a place outside of the Flick object tree. Section 2.4 explains how to test Flick against the set of included sample IDL inputs and driver programs.

2.2 Building the Flick Runtime Libraries

Flick's `configure` script automatically determines which of Flick's runtime libraries should be built and installed for your system. In general, the IIOP and ONC/TCP runtimes (for CORBA C language stubs and ONC RPC C language stubs, respectively) are always built. The other, more special-purpose runtime libraries are built only when `configure` detects that special header files are available on your system.

In particular, Flick's `configure` script looks for the following header files in order to decide which runtime libraries will be compiled on your system:

`<sys/socket.h>`	If this file is found, the IIOP and ONC/TCP runtimes will be built.
`<mach/mach_types.h>`	If this file exists, the Mach 3 runtime will be built.
`<tpz_api.h>`	If this file is found, the Trapeze runtime will be built. `configure` will look in `$(TPZ_HOME)/include` for this header.

[2]Cygwin users should read the `support/cygwin32/README` file to learn about additional support files that must be installed before Flick can be compiled on Windows 95/NT with the Cygwin library and tools.

`<gm.h>`	If this file exists, the Khazana runtime will be built. `configure` will look in `$(KHAZANA_ROOT)/src` for this file.

Flick's C++ CORBA stub generator creates stubs that work with TAO, the real-time ORB from Washington University in St. Louis. You will need a copy of TAO version 1.0 (from `http://www.cs.wustl.edu/~schmidt/TAO.html` or `ftp://ace.cs.wustl.edu/pub/ACE/`) in order to make use of Flick's optimized C++ CORBA stubs. TAO provides all the required runtime support for Flick-generated C++ stubs, and therefore, there is no library in Flick's `runtime` directory for C++ CORBA stubs.

2.3 Installing Flick

You can install the Flick binaries, libraries, and header files in a "standard" place on your system by typing the following command in the root directory of the Flick object tree (`~/obj` in our example configuration):

```
make install
```

By default, the root installation directory is `/usr/local`. You can change this default by specifying the `--prefix` option to `configure`, e.g.:

```
../flick-2.1/configure --prefix=/usr/local/flick

make            # to build Flick
make install    # to install Flick under '/usr/local/flick'
```

It is not necessary to "install" Flick in order to run the provided tests (as described in the next section) or to build your own stubs.

2.4 Testing Flick

Flick comes with a set of test IDL inputs and example programs; these are located in the `test` subdirectory of the distribution source tree. A `make` in the root of the Flick object tree will *not* run the Flick tools against these test inputs. However, running your newly compiled tools against the provided tests is easy. In a sentence: Go to the `test` subdirectory of your Flick object tree and type `make`.

2.4.1 Running Flick

You will need ~100MB of free disk space in order to run Flick against all of the provided test IDL inputs, and an *additional* ~100MB if you want to run regression tests. To run regression tests you must acquire the "known good" output files from `ftp://flux.cs.utah.edu/flux/flick/2.1/flick-2.1-tests.tar.gz`. Download that file and unpack it *on top of your Flick source tree*. For example, if the root of your Flick source tree is `~/flick-2.1`, you would install the regression test files by typing:

```
cd ~
tar xf flick-2.1-tests.tar      # Adds to '~/flick-2.1'
```

Once you have found enough disk space and (optionally) installed the files for regression testing, go into the `test` directory of your Flick *object tree* (`~/obj/test` in our running example) and type one of the following commands:

`make`	... if you simply want to run all of the Flick compiler components. The generated '`.c`'/'`.cc`' and '`.h`' files will be output into the `pbe/*` subdirectories of `test`. If everything runs smoothly, Flick is most likely working correctly. (You should ignore warnings from the Trapeze back end about maximum message sizes, array sizes, and unsupported features.)
`make verify`	... if you want to run all of the Flick compiler components *and* compare their outputs to "known good archetype" copies of the output files (i.e., do regression testing).[3]

2.4.2 Compiling and Running the Test Programs

The above commands will produce the stub source code files, but will not attempt to compile that code or link it with client or server application code.

The test programs in `test/programs/*` illustrate how client and server code can use Flick's IIOP, ONC/TCP, Mach 3, and Trapeze runtime libraries in conjunction with Flick-generated stubs. To compile and link the programs in one of these directories, you must first compile the corresponding runtime library and then create the stubs that will be compiled and linked to create the various client and server programs. Once those steps are complete, you can navigate to the `test` directory and `make` the directory that you want to compile, e.g.:

```
make programs/iiop
```

That command will compile and link the programs that exercise the IIOP stubs. You may navigate into the `programs/iiop` directory in order to run the newly compiled test programs. Refer to Chapter 4 for lists of the available command line options.

2.4.3 Additional Tests and Test Programs

The `README` file in Flick's `test` source directory contains additional, more detailed instructions for compiling the various test IDL inputs and programs.

The `test/examples/phone` directory contains the source code for the example application programs developed in Chapter 5. All of the other example application programs in `test/examples`, however, are out of date and may not compile at all.

[3]The regression tests are slightly different on Windows platforms because a text file line break consists of a CR/LF pair rather than just LF as on UNIX systems. Thus, the generated code files will differ, and the verification will fail. However, one can still perform the regression tests provided that the comparison of text files is done in text mode. The comparison program can be overridden by supplying a definition for `CMP` on the command line as follows: `make CMP=diff verify`.

Chapter 3

Using Flick

3.1 Introduction

Flick works in three phases, each implemented as a separate program: the front end, the presentation generator, and the back end. A Flick front end is simply a parser that translates some IDL input into an intermediate representation called an AOI (Abstract Object Interface, '`.aoi`') file. Next, a "presentation generator" determines how the constructs in the AOI file (the parsed IDL file) are to be mapped onto type and function definitions in the C or C++ programming language. In other words, the presentation generator determines how the generated stubs will appear — e.g., the stubs' function prototypes — and how they will interact with user code — e.g., the stubs' parameter passing conventions. This information about the stubs is written into a PRES_C (Presentation in C/C++, '`.prc`') file. The final phase of compilation, the back end, reads a PRES_C file and produces the C or C++ code for the optimized stubs, written to use a particular transport system (e.g., TCP) and message format.

Flick has multiple implementations of each of the three phases described above. Flick has three separate front ends: one to parse CORBA IDL files, one to parse ONC RPC (Sun) IDL files, and one to parse MIG IDL files. There are five different presentation generators, implementing the CORBA (C and C++), ONC RPC, MIG, and Fluke[1] mappings of IDL constructs onto C or C++. Finally, there are seven separate back ends for producing stubs that use IIOP (C and C++), ONC/TCP, Mach messages, Trapeze,[2] Khazana-style messages,[3] and Fluke IPC.

3.1.1 Flick Programs

Every compiler pass is implemented as a separate program. Once Flick has been built, links to these programs will be located in the `bin` subdirectory of your Flick build tree:

Front Ends

`flick-fe-newcorba` The CORBA front end. Parses a CORBA IDL ('`.idl`') file and outputs an AOI file ('`.aoi`') to be read by the `flick-c-pfe-corba`,

[1] Fluke is the operating system being developed by the Flux Project at the University of Utah. Flick produces all of the stub code for Fluke. For more information, refer to `http://www.cs.utah.edu/flux/`.

[2] Trapeze is the fast Myrinet-based networking system from Duke University. For more information, see `http://www.cs.duke.edu/ari/trapeze/`.

[3] Khazana is distributed global memory service in development at the University of Utah. For more information, see `http://www.cs.utah.edu/flux/`.

flick-c-pfe-corbaxx, or flick-c-pfe-fluke presentation generators.[4]

`flick-fe-sun` — The ONC RPC front end. Parses an ONC RPC IDL (‘.x’) file and outputs an AOI file to be read by the flick-c-pfe-sun presentation generator.

`flick-c-fe-mig` — The MIG front end/presentation generator. Parses a MIG IDL (‘.defs’) file and outputs a PRES_C file to be read by any back end. Note that flick-c-fe-mig is unique because it implements both the front end and presentation generation passes. These passes are combined because MIG IDL allows users to specify presentation attributes in addition to simply defining interfaces.

Presentation Generators

`flick-c-pfe-corba` — The CORBA C language presentation generator. Reads an AOI file generated by the CORBA front end and outputs a PRES_C (‘.prc’) file describing C stubs to be read by any C language back end (i.e., any back end except flick-c-pbe-iiopxx).

`flick-c-pfe-corbaxx` — The CORBA C++ presentation generator. Reads an AOI file generated by the CORBA front end and outputs a PRES_C (‘.prc’) file describing C++ stubs to be read by any C++ language back end (as of this writing, only flick-c-pbe-iiopxx).

`flick-c-pfe-sun` — The ONC RPC presentation generator. Reads an AOI file generated by the ONC RPC front end and outputs a PRES_C file describing C stubs to be read by any C language back end.

`flick-c-pfe-fluke` — The Fluke presentation generator, used with the Fluke operating system. Reads an AOI file generated by the CORBA front end and outputs a PRES_C file describing C stubs to be read by any C language back end.

Back Ends

`flick-c-pbe-iiop` — The IIOP back end for C code. Reads a PRES_C file and outputs either client stubs or server skeletons for use with Flick’s IIOP runtime. The generated stubs use the IIOP transport and CDR data encoding.

`flick-c-pbe-iiopxx` — The IIOP back end for C++ stubs. Reads a PRES_C file and outputs either client stubs or server skeletons for use with TAO. The generated stubs use the IIOP transport and CDR data encoding.

`flick-c-pbe-sun` — The ONC/TCP back end for C stubs. Reads a PRES_C file and outputs either client stubs or server skeletons for use with Flick’s ONC/TCP runtime. The generated stubs use the ONC/TCP transport and XDR data encoding.

[4] Section 3.3.2 describes the dependencies between different front ends and presentation generators.

`flick-c-pbe-mach3mig`	The Mach 3/MIG back end for C stubs. Reads a PRES_C file and outputs either client stubs or server skeletons for use with Mach. The generated stubs use Mach IPC and MIG-style typed data encoding.
`flick-c-pbe-trapeze`	The Trapeze back end for C stubs. Reads a PRES_C file and outputs either client stubs or server skeletons for use with Trapeze. The generated stubs use Trapeze messages and either CDR or XDR data encoding, depending on where the PRES_C file originated. (XDR is used for Sun-style stubs; CDR is used for CORBA and other stub styles.)
`flick-c-pbe-khazana`	The Khazana back end for C stubs. Reads a PRES_C file and outputs either client stubs or server skeletons for use within Khazana. The generated stubs use TCP and a CDR-like data encoding.
`flick-c-pbe-fluke`	The Fluke back end for C stubs. Reads a PRES_C file and outputs either client stubs or server skeletons for use with Fluke. The generated stubs use Fluke IPC and a CDR-like data encoding.

Other Tools

`flick-aoid`	The AOI dump utility. Reads an AOI file and outputs a human-readable dump of the parse tree (an '`.aod`' file). *This program is used primarily to debug Flick.*
`flick-c-presd`	The PRES_C dump utility. Reads a PRES_C file and outputs a human-readable dump of the interface presentation (a '`.prd`' file). *This program is used primarily to debug Flick.*

3.1.2 Common Command Line Options

All Flick programs support these command line arguments and options:

input	Read from the named *input* file. Every Flick program expects to receive the name of its input file on the command line. If no input file is specified, input is taken from `stdin`.
`-o` *filename* or `--output` *filename*	Write output to the specified file. If no output file is specified, the name of the output file will be constructed by concatenating the base name of the input file with a new, appropriate extension; e.g., if a presentation generator was told to read "`foo.aoi`", the name of the output file would be "`foo.prc`". If no input file is specified on the command line, output will be written to `stdout`.
`-u` or `--usage` or `-?` or `--help`	Print a synopsis of all available flags and quit.
`-v` or `--version`	Print the version of the program and quit.

Warning: Flick's AOI and PRES_C file formats are binary file formats, not text file formats. Windows NT and Windows 95 distinguish binary streams from text streams, and set `stdin` and `stdout` to be text streams. Thus, under Windows, it is not possible to pipe AOI and PRES_C data between Flick compiler passes.

3.2 Front Ends

Flick's three front ends translate the CORBA, ONC RPC, and MIG IDLs into intermediate formats used as input to later stages of the compiler. The simplest way to use any of these front ends is to specify the IDL file on the command line:

`flick-fe-newcorba` *`foo.idl`*	Process the CORBA IDL file *`foo.idl`* to produce *`foo.aoi`*. The output AOI file can be input to the CORBA or Fluke presentation generators.
`flick-fe-sun` *`foo.x`*	Process the ONC RPC IDL file *`foo.x`* to produce *`foo.aoi`*. The resultant AOI file can be input to the ONC RPC presentation generator.
`flick-c-fe-mig` *`foo.defs`*	Process the MIG IDL file *`foo.defs`* to produce *`foo.prc`*. The MIG front end is special in that it produces a PRES_C file, not an AOI file.

3.2.1 Command Line Options

In addition to the standard flags supported by all Flick programs, the CORBA and ONC RPC front ends support the following two, mutually exclusive, flags:

`-c` *`flags`* or `--cppflags` *`flags`*	Pass *`flags`* to the C preprocessor. By default, all IDL files are piped through the C preprocessor before being parsed.
`-n` or `--nocpp`	Skip the C preprocessor and pass the IDL input directly to the front end.

The MIG front end supports the following flags in addition to the standard flags listed in Section 3.1.2:

`-I`*`dir`*	Specify an include directory for the C preprocessor.
`-D`*`sym`*	Pass *`sym`* on to the C preprocessor. A value may be specified, e.g., `-DDEBUG=2`. The default value for *`sym`* is 1.
`-U`*`sym`*	Undefine symbol *`sym`* in the C preprocessor.
`-Xcpp` *`opt`*	Pass the command line option *`opt`* to the C preprocessor.
`-c` or `--client`	Generate client stubs only. The default is to generate both client stubs and server skeletons. This option cannot be combined with `-s`.
`-s` or `--server`	Generate server skeletons only. This option cannot be combined with `-c`.

3.2.2 Additional Comments

CORBA

The CORBA front end does not currently support certain CORBA data types (`wchar`, `wstring`, `fixed`, and `native` types) nor does it support CORBA contexts. It does, however, support the 64-bit "`long long`" and "`unsigned long long`" types.

ONC RPC

The ONC RPC front end allows only one argument to be passed to each operation. This conforms to the older specification of the Sun RPC language. (RFC 1831, which defines the "modern" ONC RPC IDL, allows for multiple arguments and in-argument-list type specifications.) Further, Flick's ONC RPC front end does not recognize the keywords `hyper` and `quadruple` that are described in RFC 1832. Flick's ONC RPC front end does not support the '`%`' directive that is offered by Sun's `rpcgen` IDL compiler.

MIG

The MIG front end/presentation generator is nearly complete but does not yet support all of the MIGisms you may love (or hate). The list of not-yet-implemented features is contained in the `doc/BUGS` file in the Flick source tree.

3.3 Presentation Generators

Flick has four different presentation generators: one to produce CORBA-style C language stubs, a second to produce CORBA-style C++ stubs, a third to produce ONC RPC (`rpcgen`-style) stubs, and a fourth to produce special stubs for use in Utah's Fluke operating system. (As previously mentioned, Flick's MIG-style presentation generator is conjoined with the MIG front end.) A presentation generator can be invoked to produce either the client-side stubs or the server-side dispatch function, but not both at the same time. Simply run a presentation generator twice, with different options, to produce both files; for example:

```
flick-c-pfe-corba -c -o foo-client.prc foo.aoi     # make client
flick-c-pfe-corba -s -o foo-server.prc foo.aoi     # make server
```

3.3.1 Command Line Options

Each presentation generator accepts the following options in addition to those described in Section 3.1.2:

`-c` or `--client`	Generate client stubs. This is the default if neither `-c` nor `-s` is specified. This option cannot be combined with `-s`.
`-s` or `--server`	Generate server skeletons. This option cannot be combined with `-c`.
`-a` or `--async_stubs`	Generate nonstandard "decomposed" stubs instead of normal, synchronous RPC stubs. This experimental feature is currently implemented only in the CORBA C language presentation generator.[5]

In addition, there are many, many options for changing the `printf`-like format strings that the presentation generator uses to produce the names of various presentation elements. (Use `--usage` to see them all.) These name format options exist only for short-term use; all of these options will eventually be replaced by a more general presentation-modification facility.

[5] Decomposed stubs are not currently described in this document. For more information about Flick's support for asynchronous, decomposed stubs, see Eide et al., "Flexible and Optimized IDL Compilation for Distributed Applications," published in the Proceedings of the LCR '98 workshop and available from the Flick Web pages at `http://www.cs.utah.edu/flux/flick/`.

3.3.2 Additional Comments

General

There are still some unwarranted dependencies between certain front ends and presentation generators. Although the CORBA and ONC RPC front ends both produce AOI files, the CORBA and ONC RPC presentation generators are still "tuned" to the AOI constructs generated by the respective front ends. This means that at the moment, the CORBA and Fluke presentation generators can input only AOI files generated by the CORBA front end, and similarly, that the ONC RPC presentation generator can input only AOI files generated by the ONC RPC front end. This will eventually be corrected.

CORBA

The CORBA C language presentation generator does not produce CORBA-style allocator functions for constructed data types (as described in Section 14.8 of the CORBA 2.0 specification). Flick's CORBA runtime does provide a generic `CORBA_alloc` function, however, which takes a single argument indicating the number of bytes to be allocated.

3.4 Back Ends

Flick has seven separate, optimizing back ends to produce stubs that use IIOP (for C and C++ stubs), ONC/TCP, Mach 3 typed messages, Trapeze messages, Khazana-style messages, or Fluke IPC. As described in Section 3.3, Flick separates the client- and server-side presentations into two PRES_C files. This means that in order to generate both client and server code, you must run a back end twice: once on the client-side '`.prc`' file and once on the server-side '`.prc`' file. For instance, the two commands:

```
flick-c-pbe-iiop foo-client.prc     # make client
flick-c-pbe-iiop foo-server.prc     # make server
```

produce the client stubs (`foo-client.c`, `foo-client.h`) and then the server skeletons (`foo-server.c`, `foo-server.h`) using Flick's IIOP back end. Of course, it is important to use the same back end to produce both client and server code.

3.4.1 Command Line Options

All back ends support the following options in addition to the standard flags:

`-h` *filename* or `--header` *filename*	Output the header file to *filename*. If this is not specified, the name of the header file will be constructed from the name of the output '`.c`'/'`.cc`' file (which is specified with `-o`). If the output C/C++ file is `stdout`, the header will be inlined into the output.
`-p` *prefix* or `--prefix` *prefix*	Use *prefix* when including the generated header file from the generated C or C++ file, i.e.: `#include "`*prefix*`header.h"`.
`-s` or `--system_header`	In the output C or C++ file, include the generated header as a system header rather than a user header, i.e., using `<>` rather than `""`.
`-n` *filename* or `--inline` *filename*	Allow "simple" presentation implementation functions to be inlined. This option allows us to support fast inline functions similar to those from TAO's IDL compiler. These functions are placed in *filename*,

and included by the generated header or source depending on the compile-time directive `__ACE_INLINE__`.

`--no_timestamp`
: Do not output timestamp information into the generated header and C/C++ code files. This option is useful for regression testing.

`-i` or `--no_included_implementations`
: Suppress the definition of stubs that are derived from interfaces that were `#included` by the IDL file that was processed by the front end. *This option must be used with caution, as you may unintentionally cause Flick to omit functions that you need in order to compile your client or server application.*

`-d` or `--no_included_declarations`
: Suppress the declaration of stubs that are derived from interfaces that were `#included` by the IDL file that was processed by the front end. *This option must also be used with caution, as you may unintentionally cause Flick to omit function declarations that you need in order to compile your client or server application.*

`-m` or `--all_mu_stubs`
: Generate all marshal and unmarshal stubs. Normally Flick generates only those that are actually used by the generated code. *This option is useful mainly for debugging. The generated code is significantly larger if all marshal and unmarshal stubs are produced — generally too large for practical use.*

`-P` *`filename`* or `--presentation_implementation` *`filename`*
: Read the "presentation implementation" from *`filename`*. The presentation implementation (a '`.scml`' — Source Code Markup Language — file) describes how certain functions are to be created and written into the generated code. This option is normally not required because each back end has its own built-in default.

`--nostdinc`
: Tell the back end not to search in the standard locations for SCML files. Again, this option is generally not used, unless one is implementing or testing Flick.

`-I` *`directory`* or `--include` *`directory`*
: Look in *`directory`* for presentation implementation ('`.scml`') files. This option may be repeated in order to define a search path. Like `-P`, this option is normally not required because each back end automatically looks in the Flick installation directories for the SCML files it needs.

`-f` *`filename`* or `--file` *`filename`*
: In the output C or C++ file, output: `#include "`*`filename`*`"`. This option is often used with the IIOP C++ back end; see Section 3.4.2 for more information.

`-F` *`filename`* or `--File` *`filename`*
: In the output header file, output: `#include "`*`filename`*`"`. This option is often used with the IIOP C++ back end; see Section 3.4.2 for more information.

`-D` *`vardef`* or `--define` *`vardef`*
: Use the variable definition *`vardef`* when reading the presentation implementation (SCML) files. This option may be repeated in order to define multiple variables.

3.4.2 Additional Comments

IIOP/C++ Back End Issues

The IIOP back end for C++, `flick-c-pbe-iiopxx`, is careful to generate client and server code that can be compiled into a single program (in order to support TAO "collocated" objects). However, a side effect is that the generated C++ files for client and server each need to `#include` the other's header file in order to compile. So, the IIOP back end for C++ is normally invoked like this:

```
flick-c-pbe-iiopxx -h client.h -f server.h ...    # make client
flick-c-pbe-iiopxx -h server.h -F client.h ...    # make server
```

where `client.h` and `server.h` are replaced with your names for the client and server header files, respectively.

The first command generates the client code and header files; the client's header file is named `client.h` and the generated client C++ file will `#include "server.h"`. Similarly, the second command generates the server code and header files; the server's header file is named `server.h` and the server's header will `#include "client.h"`. Note that one uses `-f` when generating the client and `-F` when generating the server. This is required in order to sequence the `#include`s properly.

Code Generation Issues

All of Flick's back ends produce code that is optimized for speed, and Flick is currently single-minded in pursuit of fast code. The current version of Flick largely overlooks other code generation issues that may be important to certain application domains:

Code Size. Because Flick aggressively inlines code to marshal and unmarshal data, the size of the generated stub functions can be very large. The generated code becomes larger as the complexity of the IDL-defined interfaces increases. A future release of Flick will make better inlining decisions.

Code Complexity. Flick-generated code is not intended to be human-readable, although the code is generally not too difficult for programmers to understand. Flick makes extensive use of the C preprocessor. Some compilers may have problems with Flick-generated code; for instance, we discovered that Microsoft Visual C++ 4.2 was unable to deal with deeply nested `switch` statements.

Thread Safety. Flick does not generate thread-safe code. If your application is multithreaded, you will need to provide explicit locks around the Flick-generated stubs.

Error Handling. Although the generated stubs generally handle errors in an appropriate manner, Flick's runtimes may fail an `assert` or `exit` in response to certain critical errors. A future release of Flick will better address error handling and recovery within the runtimes.

The `doc/BUGS` file in the Flick source tree lists known bugs with Flick's code generators.

Server Main Functions

The IIOP (C only), ONC/TCP, Khazana, and Trapeze back ends each produce a `main` function for the server, whereas the IIOP C++, Mach 3, and Fluke back ends do not. The assumption is that the former back ends will be used to generate "standalone" servers while the latter back ends will generate code for use in a more complex framework.

Mix-and-Match Problems

While it is possible to combine presentation generators and back ends in unusual ways — e.g., create Sun-style ONC/TCP code for a CORBA-style presentation — Flick's runtime libraries may not be able to handle unusual combinations. In this release we have focused on code generation for "matched" presentation and code generation, meaning:

- CORBA-style C stubs using IIOP (`flick-c-pfe-corba` with `flick-c-pbe-iiop`);
- CORBA-style C++ stubs using IIOP (`flick-c-pfe-corbaxx` with `flick-c-pbe-iiopxx`);
- ONC RPC-style stubs using ONC/TCP (`flick-c-pfe-sun` with `flick-c-pbe-sun`);
- MIG-style stubs using Mach 3 (`flick-c-fe-mig` with `flick-c-pbe-mach3mig`); and
- Fluke-style stubs using Fluke IPC (`flick-c-pfe-fluke` with `flick-c-pbe-fluke`).

These "mixed" combinations have also been tested and should work:

- CORBA-style stubs using Mach 3 (`flick-c-pfe-corba` with `flick-c-pbe-mach3mig`)[6];
- CORBA-style stubs using Trapeze (`flick-c-pfe-corba` with `flick-c-pbe-trapeze`); and
- ONC RPC-style stubs using Trapeze (`flick-c-pfe-sun` with `flick-c-pbe-trapeze`).

[6] Although this combination works, there are several serious message encoding issues to be resolved: most notably, the encodings of interface and operation identifiers.

Chapter 4

Using Flick's Runtime Libraries

Flick's runtimes provide the definitions and functions that are necessary in order for you to compile Flick-generated stubs into working code. For instance, Flick's runtime header files provide the definitions of the macros used in generated stubs. Flick's runtime libraries provide functions to manage object references, interact with the underlying IPC layer, and so on.

Flick's current runtimes were written to provide the minimum functionality required by Flick-generated code. The runtimes are not intended to be highly flexible or composable or even complete; for instance, Flick's IIOP runtime for C stubs has minimal CORBA ORB functionality. The current set of runtimes is simply sufficient to get clients and servers talking to one another. Future versions of Flick will include more complete runtimes, or will include more code generators that work with runtimes developed by third parties.

4.1 The Runtime Libraries and Headers

Flick's runtime libraries are contained under the `runtime/libraries` directory of the Flick distribution:

`libflick-iiop.a` — The IIOP runtime library for C stubs, to be using with stubs generated by Flick's IIOP back end, `flick-c-pbe-iiop`.

`libflick-suntcp.a` — The ONC/TCP runtime library, to be used with stubs generated by Flick's ONC/TCP back end, `flick-c-pbe-sun`.

`libflick-mach3mig.a` — The Mach 3 runtime library, to be used with stubs generated by Flick's Mach 3 back end, `flick-c-pbe-mach3mig`.

`libflick-trapeze.a` — The Trapeze runtime library, to be used with stubs generated by Flick's Trapeze back end, `flick-c-pbe-trapeze`.

The runtime libraries for three of Flick's back ends are distributed separately and are not further described in this manual:

`flick-c-pbe-iiopxx` — As previously described, this back end generates C++ stubs that work with TAO version 1.0, the real-time ORB from Washington University in St. Louis. You will need a copy of TAO (from `http://www.cs.wustl.edu/~schmidt/TAO.html` or `ftp://ace.cs.wustl.edu/pub/ACE/`) in order to run the code produced by Flick's IIOP/C++ back end.

`flick-c-pbe-khazana`	The Khazana runtime will be included as part of the Khazana distribution. See `http://www.cs.utah.edu/flux/` for information about Khazana.
`flick-c-pbe-fluke`	The runtime library and header files for Fluke stubs are included as part of the Fluke operating system. See `http://www.cs.utah.edu/flux/` for information about Fluke.

Flick's runtime header files are contained in the `runtime/headers` directories of your Flick source tree *and* your Flick build tree. While most of Flick's runtime headers are located in the Flick source tree, two of Flick's runtime header files are generated by Flick's `configure` script, and are therefore located in the *object tree* that you create when you build the Flick tools. To compile Flick-generated stubs, you must make sure that both the source- and object-tree `runtime/headers` directories are contained in your C compiler's system include path. Alternately, the Flick installation process (described in Section 2.3) will place all of the required header files in a single place on your system.

The rest of this chapter describes Flick's IIOP and ONC/TCP runtimes in detail. Chapter 5 shows how to use each of these runtimes in a simple phonebook application.

4.2 The IIOP Runtime

The IIOP runtime is a *very minimal* ORB-like system, implementing only the functions that are required to implement basic client/server communication. The current runtime should be sufficient, however, to allow Flick-generated stubs to communicate with any other implementation of IIOP, including programs running on top of other ORBs. The following sections describe how to use Flick's IIOP runtime. A complete application based on the IIOP runtime is described in Section 5.1.

4.2.1 Server Command Line Options

When an IIOP server starts, the Flick-generated `main` function calls `CORBA_ORB_init` to initialize the IIOP runtime and start the ORB. Flick's runtime searches for and processes the following command line arguments:

`-ORBipaddr` *`ipaddress`*	(Optional.) Specify the network interface for the server. This option is useful if your machine has multiple network interfaces. If unspecified, the server will use the host's primary network interface.
`-ORBid` *`name`*	(Optional.) Use *`name`* as the server's ORB identifier; see Section 14.26.1 of the CORBA 2.0 specification. This option is generally not used. The default ORB identifier is "FlickORB--".
`-OAport` *`portnumber`*	(Required.) Specify the port that the server will use. There is no default value; the port number must be explicitly given on the command line.
`-OAid` *`name`*	(Optional.) Use *`name`* as the server's BOA (Basic Object Adapter) identifier; see Section 14.26.2 of the CORBA 2.0 specification. Like `-ORBid`, this option is rarely used. The default BOA identifier is "FlickBOA--".

4.2.2 Creating Object Implementations

After initializing the ORB and the BOA (done for you by the Flick-generated `main` function), a server must create one or more object implementations (object instances) that will be available to service client requests. These implementations are created by a function called `register_objects` which you must write as part of your server's application code. The prototype for this function is:

```
void register_objects(CORBA_ORB, CORBA_BOA, int argc, char **argv,
                      CORBA_Environment *);
```

The `main` function generated by Flick will call your version of `register_objects` in order to initialize your server's objects. In general, your `register_objects` function should parse the given command line in order to create its objects. Your function should call `CORBA_BOA_create` to create each of its object instances:[1]

```
CORBA_Object CORBA_BOA_create(CORBA_BOA boa,
                              CORBA_ReferenceData *obj_key,
                              const char *obj_type, FLICK_SERVER obj_impl,
                              CORBA_Environment *ev);
```

The first argument should be the `CORBA_BOA` that was passed to your `register_objects` function. The second argument is a pointer to an octet sequence that names the new object instance. This name must be unique across all object instances. As is usual, the sequence is described by a C structure containing three fields: `_buffer`, `_length`, and `_maximum`. The third argument is a string that identifies the type (class) of the object; this string may be in any format you choose. (Flick does not parse the object type string; Flick uses the string only when creating and parsing object references.) The fourth argument to `CORBA_BOA_create` is a pointer to the Flick-generated dispatch function for the object interface; as far as Flick's IIOP runtime is concerned, this is what determines the object's class. The name of the server dispatch function is usually the name of the interface concatenated with the suffix "`_server`". The fifth and final argument is a pointer to a `CORBA_Environment`; your `register_objects` function should check the environment after each call to `CORBA_BOA_create` to see if an error has occurred.

It is useful for your `register_objects` function to create any additional data structures that your server will need in order to represent its objects. For instance, the phonebook application in Section 5.1 creates a list of structures to represent its phonebooks; each structure is keyed by a `CORBA_ReferenceData` and contains the data for a single phonebook object. The server work functions later use `CORBA_BOA_get_id` to locate the data for a phonebook object, mapping from a CORBA object reference to the object's `CORBA_ReferenceData`, and then using that name to locate the phonebook object's data.

Your `register_objects` function should raise an exception if it is unable to initialize your server's objects. If no exception is raised, then the Flick-generated `main` server function will begin to process requests on the registered objects.

4.2.3 Creating Object References

A client can create an object reference with the following function:

[1] Flick's version of `CORBA_BOA_create` differs slightly from what is specified by Section 8.2 of the CORBA 2.0 specification. This will eventually be fixed, probably by updating the runtime to conform to the newer Portable Object Adapter (POA) specification.

```
CORBA_Object CORBA_ORB_string_to_object(CORBA_ORB orb, CORBA_char *name,
                                        CORBA_Environment *ev);
```

The name can be either a stringified Interoperable Object Reference (IOR) or a URL-style name of the form "`iiop:1.0//`*`hostname`*`:`*`port`*`/`*`objecttype`*`/`*`objectname`*".[2] The latter form is generally preferred because it is easier to type. Whenever a Flick-based server creates an object instance, the server prints both the IOR and URL-style name of the new object.

Alternately, a client can create an object reference by going through the following steps:

1. Initialize an ORB, using `CORBA_ORB_init`.
2. Create a BOA, using `CORBA_ORB_BOA_init`.
3. Create the object reference by calling `CORBA_BOA_create` with the appropriate `CORBA_ReferenceData` (key) value.

This procedure will find the corresponding object instance within the given ORB and create an appropriate object reference. Of course, `CORBA_ORB_string_to_object` is generally simpler for clients to use.

4.2.4 Using Object References

Once a client has obtained an object reference, it may use the reference to invoke operations on the object as defined by the object's IDL interface. Flick's IIOP runtime provides these additional functions for manipulating object references:

`CORBA_char *CORBA_ORB_object_to_string(CORBA_ORB, CORBA_Object, CORBA_Environment *);`
Create and return a stringified Interoperable Object Reference (IOR).

`CORBA_char *CORBA_ORB_object_to_readable_string(CORBA_ORB, CORBA_Object, CORBA_Environment *);`
Create and return a human-readable, URL-style stringified object reference. (This is not a standard CORBA function.)

`CORBA_Object CORBA_ORB_string_to_object(CORBA_ORB, CORBA_char *, CORBA_Environment *);`
Create and return an object reference from a stringified object reference (IOR or URL).

`CORBA_ReferenceData *CORBA_BOA_get_id(CORBA_BOA, CORBA_Object, CORBA_Environment *);`
Return the corresponding key for the given object reference.

`CORBA_boolean CORBA_Object_is_nil(CORBA_Object, CORBA_Environment *);`
Return true if the object reference is nil.

`CORBA_Object CORBA_Object_duplicate(CORBA_Object, CORBA_Environment *);`
Duplicate an object reference.

`void CORBA_Object_release(CORBA_Object, CORBA_Environment *);`
Release an object reference.

`CORBA_unsigned_long CORBA_Object_hash(CORBA_Object, CORBA_unsigned_long, CORBA_Environment *);`
Return a value suitable for use in a hash table. The value is not guaranteed unique, but good enough to implement an efficient hash table with the number of buckets indicated by the second argument.

`CORBA_boolean CORBA_Object_is_equivalent(CORBA_Object, CORBA_Object, CORBA_Environment *);`
Return true if the two object references refer to the same object instance.

[2] The idea for URL-style object references was adopted from SunSoft's reference IIOP implementation, which is available at `ftp://ftp.omg.org/pub/interop/iiop.tar.Z`. Flick's URL-style references differ from those supported by the SunSoft IIOP implementation in that Flick's references include the object type.

4.3 The ONC/TCP Runtime

Flick's ONC/TCP runtime was designed so that it would be possible for users to write an ONC RPC-based client or server, and then compile the application code to use either Flick-generated stubs or `rpcgen`-generated stubs. This requirement introduced a few oddities into Flick's ONC/TCP runtime but can be very helpful in demonstrating the speed gained by using Flick-generated stubs. (If you are curious, the files `test/programs/sun/sunstat-work.c` and `test/programs/sun/sunstat-use.c` in the Flick distribution show how to write code that is compatible with both Flick and `rpcgen`.)

4.3.1 Starting a Server

Flick's ONC/TCP back end creates a `main` function that automatically registers the interfaces described in the original IDL file. (In an ONC RPC IDL file, an interface is a program-version pair.) There are no objects to create or register; the server itself is the "object" in ONC RPC. The Flick-generated `main` function then executes the server main loop, receiving requests and sending replies.

If the Flick-generated `main` function is suitable for your application, then you need do nothing more. If, however, your server needs to perform special initialization, you will have to write your own `main` function to set up your application, register your interfaces, and then run the main server loop. The rest of this section describes the procedure for registering your interfaces with Flick's ONC/TCP runtime.

Your `main` function must initialize a `FLICK_SERVER_DESCRIPTOR` structure for each interface that it wants to register:

```
typedef struct FLICK_SERVER_DESCRIPTOR {
        unsigned int prog_num;   /* The program number. */
        unsigned int vers_num;   /* The version number. */
        ...
} FLICK_SERVER_DESCRIPTOR;
```

The values of `prog_num` and `vers_num` fields would normally be the program and version numbers listed in your original IDL file. After filling in the `FLICK_SERVER_DESCRIPTOR` structure, your server must call `flick_server_register` to register the interface with the runtime:

```
/* Returns 1 for success or 0 for failure. */
int flick_server_register(FLICK_SERVER_DESCRIPTOR, FLICK_SERVER);
```

The first argument is the structure containing the program and version numbers that was previously filled in. The second argument is a pointer to the Flick-generated server dispatch function for the interface. The name of this function is generally the name of the ONC RPC program, followed by an underscore and the version number of the interface: "*`program_version`*".

After your `main` function has registered all of its interfaces, it must call the `flick_server_run` function. This function takes no arguments and should never return. If it does, there has been a fatal error.

4.3.2 Connecting to a Server

When a server is run, it registers itself with the RPC portmapper facility. Client processes can connect to a running server simply by specifying the host machine of the server and the program and version numbers of the desired interface. To connect to a server, a client process fills out a `FLICK_SERVER_LOCATION` structure:

```
typedef struct FLICK_SERVER_LOCATION {
        char *server_name;      /* The host machine.   */
        unsigned int prog_num;  /* The program number. */
        unsigned int vers_num;  /* The version number. */
} FLICK_SERVER_LOCATION;
```

Once this structure is initialized, the client calls `flick_client_create` to connect to the server:

```
/* Returns 1 for success or 0 for failure. */
int flick_client_create(CLIENT *, FLICK_SERVER_LOCATION);
```

The first argument must point to a `CLIENT` structure; that structure will be initialized by the call to `flick_client_create`.

4.3.3 Using the Server Connection

Once a client has established a connection to a server, it must pass the initialized `CLIENT` data to the Flick-generated stub functions in order to make RPCs. When the client is ready to close its connection to the server, it should call `flick_client_destroy`:

```
void flick_client_destroy(CLIENT *);
```

4.3.4 Using the Service Request Data (`svc_req`)

The second argument to an ONC RPC-style server work function is generally a pointer to a `struct svc_req`, a structure that describes the context of the current RPC: the program, version, and operation numbers, the client's credentials, and so on. When writing a server work function, you must include this argument in your function's parameter list; see Section 5.3.2 for examples.

However, beware: *Flick's ONC/TCP runtime does not currently initialize the structure that is passed to your work function!* Currently, Flick-generated stubs include this parameter simply to be prototype-compatible with `rpcgen`-generated code. In a future version of Flick, the correct `svc_req` data will be provided to server work functions.

Chapter 5

Putting It Together: A Phonebook Application

This chapter illustrates the use of Flick for implementing a simple network "phonebook." In our examples, a single server process will maintain one or more phonebooks, with each phonebook containing a set of name-and-number entries. Client processes will be able to connect to the server in order to add, remove, or locate phonebook entries. The complete phonebook application will be implemented in three different ways: first as a CORBA C program, then as a CORBA C++ program, and finally as an ONC RPC (C) program. By reading through these examples, you will learn how to use Flick to develop your own network applications.

5.1 The CORBA Phonebook in C

The source code for the CORBA phonebook application is contained in the `test/examples/phone/corba` directory of the Flick source tree.

5.1.1 Interface Definition

The CORBA IDL specification of our phonebook interface is straightforward:

```
module data {
    typedef string<200> name;
    typedef string<20>  phone;

    struct entry { name n; phone p; };

    exception duplicate { phone p; };
    exception notfound  {};

    interface phonebook {
        void  add(in entry e)   raises (duplicate);
        void  remove(in name n) raises (notfound);
        phone find(in name n)   raises (notfound);
    };
};
```

Assuming that this interface is contained in a file `phone.idl`, the following commands will compile the phonebook specification into client stubs and a server skeleton:[1]

```
flick-fe-newcorba phone.idl

flick-c-pfe-corba -c -o phone-client.prc phone.aoi
flick-c-pbe-iiop phone-client.prc
# Final outputs: 'phone-client.c' and 'phone-client.h'.

flick-c-pfe-corba -s -o phone-server.prc phone.aoi
flick-c-pbe-iiop phone-server.prc
# Final outputs: 'phone-server.c' and 'phone-server.h'.
```

5.1.2 Server Functions

For the server, Flick will create a `main` function (found in `phone-server.c`). To complete the server you must:

- define the data structures that the server will need in order to manage its objects;
- write a `register_objects` function that will instantiate the server's objects (as described in Section 4.2.2); and
- write functions to implement the operations listed in the IDL file.

Our simple server will use a linked list to represent its collection of phone book objects. Each node in that list will be a '`struct impl_node_t`' and will represent a single phonebook:

```
typedef struct impl_node_t *impl_node;

struct impl_node_t {
    CORBA_ReferenceData id;     /* The search key for this node. */
    data_phonebook obj;         /* CORBA Object ref for this.    */

    data_entry **pb;            /* The array of ptrs to entries. */
    int pb_elems;               /* # of entries in 'pb'.         */
    int pb_size;                /* The size of the 'pb' array.   */

    impl_node next;             /* Ptr to the next list element. */
};

/* 'pb_impl_list' is the list of phonebook implementations. */
impl_node pb_impl_list = 0;
```

The `CORBA_ReferenceData` slot of each node will allow us to find the data for a phonebook, given its CORBA object reference.[2] The other slots should be obvious, although you may be wondering where

[1] The actual commands issued by the `Makefile` are somewhat more complicated because they do not assume that you have installed Flick on your system. Read the `Makefile` for details.

[2] This example uses a linked list to represent the server's set of phonebooks, but a more sophisticated application would use a hash table, with keys generated by the `CORBA_Object_hash` function.

data_entry is defined. It is defined in the file phone-server.h, which is created for you by Flick. The data_entry structure type is automatically generated from the the entry structure declaration in the source IDL file.

The next step is to write register_objects, which will be called from the Flick-generated main function so that the server can instantiate its objects and register them with the Flick runtime (the ORB). The code for this function is shown below and is contained in the phone-workfuncs.c file in Flick's CORBA phonebook source directory. This register_objects function searches the command line for options of the form "-I *name*" and creates one object for each such option. In the code below, note that data_phonebook_server is the name of the server skeleton function that was generated for you by Flick. That is the function that dispatches requests to objects that support the phonebook interface defined in our IDL.

```
void register_objects(CORBA_ORB o, CORBA_BOA b, int argc, char **argv,
                      CORBA_Environment *ev)
{
    int i;

    orb = o; /* Globals: save ORB and BOA handles for later use. */
    boa = b;

    pb_impl_list = 0;

    for (i = 1; i < argc; i++) {
        if (!strcmp(argv[i], "-I") && (i + 1 < argc)) {
            CORBA_ReferenceData id;
            data_phonebook obj;
            impl_node list = pb_impl_list;

            /* Register object with name 'argv[i + 1]'. */
            id._length = strlen(argv[i + 1]);
            id._maximum = id._length;
            id._buffer = (CORBA_char *)
                          CORBA_alloc(id._maximum * sizeof(CORBA_char));
            if (!id._buffer) {
                signal_no_memory(ev);
                return;
            }
            memcpy(id._buffer, argv[i + 1], id._length);

            obj = CORBA_BOA_create(boa, &id,
                                   "data::phonebook",
                                   &data_phonebook_server,
                                   ev);
            if (ev->_major != CORBA_NO_EXCEPTION)
                return;

            /* Add a corresponding 'impl_node' to our list. */
            pb_impl_list = (impl_node)
```

```
                           malloc(sizeof(*pb_impl_list));
                if (!pb_impl_list) {
                    signal_no_memory(ev);
                    return;
                }

                pb_impl_list->id = id;
                pb_impl_list->obj = obj;
                pb_impl_list->pb = 0;
                pb_impl_list->pb_elems = 0;
                pb_impl_list->pb_size = 0;
                pb_impl_list->next = list;
            }
        }

        /* Signal failure if no objects were registered. */
        if (!pb_impl_list)
            signal_initialize(ev);
    }
```

We are now ready to write the server "work functions" — the functions that implement the operations defined in our phonebook interface. Abbreviated code for the `data_phonebook_add` function (implementing the `add` operation defined in our IDL) is shown below. Complete source code for all the work functions is contained in the `phone-workfuncs.c` file in the example source code directory.

```
void data_phonebook_add(data_phonebook obj, data_entry *arg,
                        CORBA_Environment *ev)
{
    /* Find the object implementation --- see 'find_impl' below. */
    impl_node impl = find_impl(obj, ev);
    int i;

    /* If we failed to find the implementation, return. */
    if (ev->_major != CORBA_NO_EXCEPTION)
        return;

    /* See if this entry is already in the phonebook. */
    for (i = 0; i < impl->pb_size; ++i) {
        if (impl->pb[i] && !strcmp(impl->pb[i]->n, arg->n)) {
            /* Found a duplicate!  Raise a 'data_duplicate' exception. */
            data_duplicate *d =
                (data_duplicate *) CORBA_alloc(sizeof(data_duplicate));

            if (!d) {
                signal_no_memory(ev); return;
            }
            d->p = (CORBA_char *) CORBA_alloc(strlen(impl->pb[i]->p) + 1);
            if (!d->p) {
```

```
                CORBA_free(d); signal_no_memory(ev); return;
            }
            strcpy(d->p, impl->pb[i]->p);
            CORBA_BOA_set_exception(boa, ev, CORBA_USER_EXCEPTION,
                                    ex_data_duplicate, d);
            return;
        }
    }

    /* Find an empty entry in 'impl'; grow the phonebook if necessary. */
    i = find_empty_entry(impl, ev);
    if (ev->_major != CORBA_NO_EXCEPTION)
        return;

    /*
     * Allocate memory for the new entry.  Note that we have to copy the
     * 'arg' data because CORBA says we can't keep pointers into 'in' data
     * after this function has returned.
     */
    impl->pb[i] = (data_entry *) malloc(sizeof(data_entry));
    if (!impl->pb[i]) {
        signal_no_memory(ev); return;
    }

    impl->pb[i]->n = (char *) malloc(sizeof(char) * (strlen(arg->n) + 1));
    impl->pb[i]->p = (char *) malloc(sizeof(char) * (strlen(arg->p) + 1));
    if (!(impl->pb[i]->n) || !(impl->pb[i]->p)) {
        /* Free what we have allocated and signal an exception. */
        ... ; return;
    }

    /* Copy the 'arg' information into our phonebook. */
    strcpy(impl->pb[i]->n, arg->n);
    strcpy(impl->pb[i]->p, arg->p);

    /* Increment the number of entries in our phonebook. */
    impl->pb_elems++;

    /* Success! */
    return;
}
```

The above function calls `find_impl` to find the node that represents a phonebook, given its CORBA object reference. That helper function is shown below.

```
impl_node find_impl(data_phonebook obj, CORBA_Environment *ev)
{
    impl_node this_impl = pb_impl_list;
```

```
    CORBA_ReferenceData *obj_id = CORBA_BOA_get_id(boa, obj, ev);

    if (ev->_major != CORBA_NO_EXCEPTION)
        return 0;

    while (this_impl) {
        if ((this_impl->id._length == obj_id->_length)
            && !memcmp(this_impl->id._buffer,
                       obj_id->_buffer,
                       obj_id->_length))
            break;
        this_impl = this_impl->next;
    }
    if (!this_impl) {
        /* This should never be reached. */
        signal_object_not_exist(ev);
    }

    CORBA_free(obj_id->_buffer);
    CORBA_free(obj_id);

    return this_impl;
}
```

5.1.3 Client Program

For the phonebook client, Flick creates stubs that will send requests to the server process and receive the server's replies. You must write the client's `main` function. Our client will be interactive, and its `main` function is shown below. Note that our client uses `CORBA_ORB_string_to_object` to establish its connection to an object that resides in the server. Flick's IIOP runtime will cause the server to print the names of its objects as they are registered. The argument to our client must be either the IOR-style or the URL-style name of an object in the server; refer to Section 4.2.3 for more information about object names.

```
int main(int argc, char **argv)
{
    CORBA_ORB orb = 0;
    CORBA_Environment ev;
    data_phonebook obj;
    int sel, done;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <phone obj reference>\n", argv[0]);
        exit(1);
    }

    orb = CORBA_ORB_init(&argc, argv, 0, &ev);
    if (ev._major != CORBA_NO_EXCEPTION) {
        printf("Can't initialize the ORB.\n");
```

```
        exit(1);
    }

    obj = CORBA_ORB_string_to_object(orb, argv[1], &ev);
    if (ev._major != CORBA_NO_EXCEPTION) {
        printf("Can't convert '%s' into an object reference.\n",
               argv[1]);
        exit(1);
    }

    done = 0;
    while (!done) {
        read_integer(("\n(1) Add an entry (2) Remove an entry "
                      "(3) Find a phone number (4) Exit: "),
                     &sel);
        switch(sel) {
        case 1:  add_entry(obj); break;
        case 2:  remove_entry(obj); break;
        case 3:  find_entry(obj); break;
        case 4:  done = 1; break;
        default: printf("Please enter 1, 2, 3, or 4.\n");
        }
    }
    return 0;
}
```

The client's `add_entry` function invokes `data_phonebook_add`, which is the Flick-generated stub for the `add` operation. It handles exceptions by calling the `handle_exception` function; the source for `handle_exception` can be found in the `phonebook.c` file in our example's source directory.

```
void add_entry(data_phonebook obj)
{
    data_entry e;
    char name[NAME_SIZE], phone[PHONE_SIZE];
    CORBA_Environment ev;

    e.n = name;
    e.p = phone;

    read_string("Enter the name: ", e.n, NAME_SIZE);
    read_string("Enter the phone number: ", e.p, PHONE_SIZE);

    data_phonebook_add(obj, &e, &ev);
    if (ev._major != CORBA_NO_EXCEPTION)
        handle_exception(&ev);
    else
        printf("'%s' has been added.\n", name);
}
```

5.1.4 Compiling the Application

The `test/examples/phone/corba` directory contains a simple `Makefile` for compiling the phonebook server and client programs. You will need to edit the `Makefile` slightly in order to suit your build environment. Once that is done, and you have built Flick and the IIOP runtime, you should be able to type `make` to build the CORBA phonebook. Two programs will be created: `phoneserver`, the server, and `phonebook`, the client.

5.1.5 Using the Phonebook

To run the application you must first start the phonebook server. The server expects to receive at least two arguments on the command line:

`-OAport` *portnumber*	The port on which to run the server.
`-I` *name*	The names of the object instances that the server will create. This option may be repeated in order to create several object instances. Each object must have a unique name.

See Section 4.2.1 for a list of additional, optional arguments. Once the `phoneserver` program is running, you can invoke the `phonebook` program, giving it the IOR or URL-style name of a server object.

```
1 marker:~> phoneserver -OAport 1253 -I OfficeList -I DeptList
Warning: no '-ORBipaddr' specified; using 'marker.cs.utah.edu'.
Object 'OfficeList' is ready.
  URL:  iiop:1.0//marker.cs.utah.edu:1253/data::phonebook/OfficeList
  IOR:  IOR:010000001000000646174613a3a70686f6e65626f6f6b0001000...
Object 'DeptList' is ready.
  URL:  iiop:1.0//marker.cs.utah.edu:1253/data::phonebook/DeptList
  IOR:  IOR:010000001000000646174613a3a70686f6e65626f6f6b0001000...

# Run a client on the same machine or on a different machine.

1 fast:~> phonebook iiop:1.0//marker.cs.utah.edu:1253/...

(1) Add an entry (2) Remove an entry (3) Find a phone number (4) Exit:
...
```

5.2 The CORBA Phonebook in C++

In this section we will show you how to make a CORBA C++ version of the phonebook application. The source code for this example can be found in the `test/examples/phone/tao-corbaxx` directory of the Flick distribution.

Note that you must acquire and install TAO version 1.0, the real-time ORB from Washington University in St. Louis, before you will be able to compile the CORBA C++ phonebook or any other Flick-generated CORBA C++ code. See Section 2.2 for more information.

5.2.1 Interface Definition

The CORBA IDL specification of our phonebook, contained in the `phone.idl` file, is the same as the IDL shown previously in Section 5.1.1. Because CORBA IDL is independent of the target programming language, we can use the same IDL file to generate both C and C++ versions of our phonebook application. We produce our C++ stubs with the following commands:[3]

```
flick-fe-newcorba phone.idl

flick-c-pfe-corbaxx -c -o phoneC.prc phone.aoi
flick-c-pbe-iiopxx -f phoneS.h phoneC.prc
# Final outputs: ‘phoneC.cc’ and ‘phoneC.h’.

flick-c-pfe-corbaxx -s -o phoneS.prc phone.aoi
flick-c-pbe-iiopxx -F phoneC.h phoneS.prc
# Final outputs: ‘phoneS.cc’ and ‘phoneS.h’.
```

The reasons for the `-f` and `-F` command line options are described in Section 3.4.2.

5.2.2 Server Implementation

Flick generates a server skeleton from the phonebook IDL. To complete the server, we need to write an actual implementation of the phonebook object ourselves. Writing an implementation class is just like writing a regular C++ class, but with some additional idioms imposed by CORBA. Below is the definition of our phonebook implementation class, called `phone_i`:

```
class phone_i : public POA_data::phonebook
{
public:
    phone_i();
    ~phone_i();

    /* The functions that implement the ‘phone.idl’ interface. */
    virtual void add(const data::entry &e, CORBA::Environment &);
    virtual void remove(const char *n, CORBA::Environment &);
    virtual data::phone find(const char *n, CORBA::Environment &);

private:
    /* We implement the phonebook with a simple linked list. */
    struct impl_node_t {
        data::entry         entry;
        struct impl_node_t *next;
    };
    typedef struct impl_node_t *impl_node_ptr;

    /* The head of the linked list. */
```

[3]The actual commands issued by the `Makefile` are somewhat more complicated because they do not assume that you have installed Flick on your system. Read the `Makefile` for details.

```
    impl_node_ptr head;

    /*
     * `find_node' locates and returns an `impl_node_ptr' to an existing
     * node that contains the given name, or a null pointer if no node
     * contains the name.
     */
    impl_node_ptr find_node(const char *name);
};
```

The declaration of this class is made up of a constructor and destructor, some virtual functions corresponding to the interface operations, and some private members used in the implementation.

Notice that the `phone_i` class inherits from a special class called `POA_data::phonebook`. This *POA class*, also called a *skeleton class*, is created for you by Flick and is the abstract base class for all implementations of the `data::phonebook` interface. The skeleton class takes care of marshaling and unmarshaling data for the methods that implement the phonebook interface (`add`, `remove`, and `find`). Flick also generates special "tie" and "collocated" classes for each interface in the IDL; to learn more about these, consult the CORBA specification and the TAO documentation.

Now that the `phone_i` class is declared, we need to write the methods that implement the operations listed in our IDL. Code for the `phone_i::add` method is shown below. Complete source code for all of the `phone_i` methods is contained in the `phone_i.cc` file in the example source code directory.

```
void phone_i::add(const data::entry &e, CORBA::Environment &ACE_TRY_ENV)
{
    impl_node_ptr node_ptr;

    /* Check for a duplicate name. */
    node_ptr = find_node(e.n);
    if (node_ptr) {
        /*
         * We already have this number.  Use the `ACE_THROW' macro to
         * throw an exception.
         */
        ACE_THROW(data::duplicate(node_ptr->entry.p));
    } else {
        /*
         * Make a new node and add it in.  Note that the structure
         * assignment below does a *deep* copy: the structure type is
         * defined in IDL, and the generated assignment operator does a
         * deep copy as required by the CORBA C++ language mapping.
         */
        node_ptr = new impl_node_t;
        node_ptr->entry = e; /* Deep copy. */
        node_ptr->next = head;
        head = node_ptr;
    }
}
```

The implementation of our phonebook operations is rather ordinary C++ code, plus some special idioms for handling exceptions. CORBA suggests that an IDL-to-C++ compiler should produce code that handles CORBA-defined exceptions as regular C++ exceptions, whenever the target C++ compiler has working exception support. Flick does not yet implement this code style because many C++ compilers still have poor support for exceptions. Instead, Flick supports the CORBA standard's *alternate mapping* for exceptions, which works with all C++ compilers. In this mapping, CORBA exceptions are communicated through an extra `CORBA::Environment` parameter to the generated stubs, as in the standard CORBA C language mapping. TAO provides some exception-related macros (e.g., `ACE_THROW`) that hide the details of exception handling, thus allowing our code to be largely independent of the specific exception mapping implemented by Flick. Refer to the TAO documentation for more information about these facilities.

Once the `phone_i` implementation is complete, the only remaining task for our server is to implement the `main` function. Our `main` function initializes the ORB and then creates a set of phonebook objects as described by the server's command line. (The command line is parsed by the function `parse_args`, not shown.) When everything is set up, we simply set the system in motion and let the server process requests from clients.

```
int main(int argc, char **argv)
{
    TAO_ORB_Manager orb_manager;
    CORBA::Environment ACE_TRY_ENV;

    /* Use 'try' because there might be an exception. */
    ACE_TRY {
        phone_impl *pi;

        /* Initialize the POA. */
        orb_manager.init_child_poa(argc, argv, "child_poa", ACE_TRY_ENV);
        ACE_TRY_CHECK;

        /* Parse the command line; creates 'the_impls' list. */
        if (parse_args(argc, argv) != 0)
            return 1;

        /*
         * Walk through the list of implementations and attach each
         * one to the server.
         */
        for (pi = the_impls; pi; pi = pi->next) {
            /*
             * Add the head of the 'pi' list to the POA and get the
             * IOR for the object.
             */
            CORBA::String_var ior
                = (orb_manager.
                   activate_under_child_poa(pi->name,
                                            &pi->servant,
                                            ACE_TRY_ENV));
            ACE_TRY_CHECK;
```

```
            /* Give the IOR to the user. */
            ACE_DEBUG((LM_DEBUG, "Object '%s' is ready.\n", pi->name));
            ACE_DEBUG((LM_DEBUG, "  IOR:  %s\n", ior.in()));
        }
        /*
         * Make sure there is at least one object and then start the
         * server.
         */
        if (the_impls)
            orb_manager.run(ACE_TRY_ENV);
        else
            ACE_ERROR_RETURN((LM_ERROR,
                              "No implementations were specified.\n"),
                             1);
        ACE_TRY_CHECK;
    }
    /* Catch any system exceptions. */
    ACE_CATCH (CORBA::SystemException, ex) {
        ex._tao_print_exception("System Exception");
        return 1;
    }
    ACE_ENDTRY;

    return 0;
}
```

5.2.3 Client Program

With the server side implemented, we can now create a client to allow for interactions with the phonebook. Flick creates the stubs that will send requests to the server process and receive the server's replies. A programmer, however, must write the client program's `main` function to initialize the ORB and establish the client's connection to a phonebook, i.e., obtain a reference to a phonebook object that resides in the server. The code for our sample client `main` function is shown below.

Our client establishes a connection by taking an IOR from the command line. (The arguments "`-k` *`IOR`*" are parsed by the function `parse_args`, not shown.) Once the client has established the phonebook connection, we can treat the phonebook reference as a normal C++ object, which makes it easy to write the rest of the client.

The client is interactive so there is a small event loop that prompts the user for requests. When the user decides to exit, the `main` function terminates, and the connection to the server is closed. (When we leave the `main` function, the '`_var`'-style object references are automatically destroyed, with the result being that our connection to the server is closed.)

```
int main(int argc, char **argv)
{
    CORBA::ORB_var orb;
    CORBA::Object_var object;
    data::phonebook_var pb;
```

```
    CORBA::Environment ACE_TRY_ENV;
    const char *op_name;
    int sel, done;

    /* Parse the command line; initializes `ior'. */
    if (parse_args(argc, argv) != 0)
        return 1;

    ACE_TRY {
        /* Initialize the ORB. */
        op_name = "CORBA::ORB_init";
        orb = CORBA::ORB_init(argc, argv, 0, ACE_TRY_ENV);
        ACE_TRY_CHECK;

        /* Get the object reference with the IOR. */
        op_name = "CORBA::ORB::string_to_object";
        object = orb->string_to_object(ior, ACE_TRY_ENV);
        ACE_TRY_CHECK;

        /* Narrow the object to a `data::phonebook'. */
        op_name = "data::phonebook::_narrow";
        pb = data::phonebook::_narrow(object.in(), ACE_TRY_ENV);
        ACE_TRY_CHECK;
    }
    ACE_CATCH (CORBA::Exception, ex) {
        ex._tao_print_exception(op_name);
        return 1;
    }
    ACE_ENDTRY;

    done = 0;
    while (!done) {
        read_integer(("\n(1) Add an entry (2) Remove an entry "
                      "(3) Find a phone number (4) Exit: "),
                     &sel);
        switch (sel) {
        case 1:  add_entry(pb); break;
        case 2:  remove_entry(pb); break;
        case 3:  find_entry(pb); break;
        case 4:  done = 1; break;
        default: printf("Please enter 1, 2, 3, or 4.\n");
        }
    }
    return 0;
}
```

The client's `add_entry` function invokes the `data::phonebook::add` method, which is the Flick-generated stub for the add operation. Because the stub can signal an exception, the `add_entry` function is careful to catch exceptions as shown below. (As previously described in Section 5.2.2, we use the `ACE_*` macros provided by TAO to make our code more portable across different C++ compilers and different ways of handling CORBA exceptions.)

```
void add_entry(data::phonebook_var obj)
{
    data::entry e;
    char name[NAME_SIZE], phone[PHONE_SIZE];
    CORBA::Environment ACE_TRY_ENV;

    read_string("Enter the name: ", name, NAME_SIZE);
    read_string("Enter the phone number: ", phone, PHONE_SIZE);
    /*
     * Duplicate the strings; they will be freed when ‘e’ is destroyed.
     */
    e.n = CORBA::string_dup(name);
    e.p = CORBA::string_dup(phone);

    ACE_TRY {
        obj->add(e, ACE_TRY_ENV);
        /* Check for exceptions. */
        ACE_TRY_CHECK;
        printf("‘%s’ has been added.\n", name);
        break;
    }
    ACE_CATCH(data::duplicate, ex) {
        /*
         * Catch a ‘data::duplicate’ exception.  It contains the
         * phone number that is already in the database.
         */
        printf("A user exception was raised: ");
        printf("duplicate, phone = ‘%s’.\n", (const char *) ex.p);
    }
    ACE_CATCH(CORBA::Exception, ex) {
        /* Catch all other exceptions. */
        ex._tao_print_exception("data::phonebook::add");
    }
    ACE_ENDTRY;
}
```

5.2.4 Compiling the Application

The `test/examples/phone/tao-corbaxx` directory contains a `Makefile` for compiling the phonebook server and client programs. You will need to edit the `Makefile` slightly in order to suit your build environment. Once that is done, and you have built both Flick and TAO, you should be able to type `make` to build the CORBA phonebook. Two programs will be created: `phoneserver`, the server, and `phonebook`, the client.

5.2.5 Using the Phonebook

To run the application you must first start the phonebook server. The server expects to receive at least one argument on the command line:

`-I` *name*	The names of the object instances that the server will create. This option may be repeated in order to create several object instances. Each object must have a unique name.

Once the `phoneserver` program is running, you can invoke the `phonebook` program, giving it the IOR of a server object.

```
1 marker:~> phoneserver -I OfficeList -I DeptList
Object 'DeptList' is ready.
  IOR:  IOR:010000001700000049444c3a646174612f70686f6e65626f6f6b3...
Object 'OfficeList' is ready.
  IOR:  IOR:010000001700000049444c3a646174612f70686f6e65626f6f6b3...

# Run a client on the same machine or on a different machine.

1 fast:~> phonebook -k IOR:010000001700000049444c3a646174612f7068...

(1) Add an entry (2) Remove an entry (3) Find a phone number (4) Exit:
...
```

5.3 The ONC RPC Phonebook

In the following sections we will reimplement our phonebook application using Flick's ONC RPC tools rather than Flick's CORBA tools. The source code for the ONC RPC phonebook is contained in the `test/examples/phone/oncrpc` directory of the Flick distribution.

5.3.1 Interface Definition

Again, the ONC RPC IDL specification of the phonebook interface is straightforward:

```
typedef string name<200>;
typedef string phone<20>;

struct entry { name n; phone p; };

program netphone {
    version firstphone {
        int     add(entry)    = 1;
        int     remove(name) = 2;
        phone   find(name)    = 3;
    } = 1;
} = 58239;
```

If you compare this interface to the one we previously defined using the CORBA IDL (see Section 5.1.1), you will notice some minor differences. Because ONC RPC does not allow us to define explicit exceptions, the return types of the `add` and `remove` operations have changed in order to indicate success or failure. Rather than change the return type of the `find` operation, however, our application will simply use an empty phone number string to indicate search failures.

Assuming that the above interface is contained in a file `phone.x`, the following commands will compile the phonebook specification into client stubs and a server skeleton:[4]

```
flick-fe-sun phone.x

flick-c-pfe-sun -c -o phone-client.prc phone.aoi
flick-c-pbe-sun phone-client.prc
# Final outputs: 'phone-client.c' and 'phone-client.h'.

flick-c-pfe-sun -s -o phone-server.prc phone.aoi
flick-c-pbe-sun phone-server.prc
# Final outputs: 'phone-server.c' and 'phone-server.h'.
```

5.3.2 Server Functions

For the server, Flick will create a `main` function (found in `phone-server.c`). In order to complete the server, you must:

- define the data structures that the server will need in order to manage its phonebook; and
- write functions to implement the operations listed in the IDL file.

In comparison with the CORBA version of our phonebook server, the ONC RPC server will be simpler because it will only manage one phonebook, not several. (If we wanted our ONC RPC server to manage multiple books, we would need to change our IDL file so that we could pass a phonebook name to each operation.) So, our ONC RPC server simply needs to maintain a single array of phonebook entries:

```
entry **pb = 0;      /* The array of pointers to entries. */
int pb_elems = 0;    /* # of entries in 'pb'.             */
int pb_size = 0;     /* The size of the 'pb' array.       */
```

The `entry` type definition is created by Flick from the `entry` definition in the original IDL file. Now we are ready to write the server work functions to implement the operations defined in our phonebook interface. The C prototypes of those functions are shown below:

`int *add_1(entry *arg, struct svc_req *obj);`
Add an entry to the phonebook. Return pointer to zero to indicate success, or pointer to non-zero to indicate an error.

`int *remove_1(name *arg, struct svc_req *obj);`
Remove the entry containing the given name from the phonebook. Return pointer to zero to indicate success, or pointer to non-zero to indicate an error.

[4] The actual commands issued by the `Makefile` are somewhat more complicated because they do not assume that you have installed Flick on your system. Read the `Makefile` for details.

`phone *find_1(name *arg, struct svc_req *obj);`
Find the phone number corresponding to the given name in the phonebook. Return an empty string to indicate failure.

The name of each function is constructed from the name of the corresponding operation defined in IDL, followed by an underscore and the version number of the interface (in this case, 1). The second argument to each function is a "service request" structure that would generally contain data about the current request and the requesting client. *Flick's ONC/TCP runtime does not currently initialize this structure!* Flick-generated stubs include this parameter simply to be prototype-compatible with `rpcgen`-generated code.

Complete code for all of these functions is contained in the `phone-workfuncs.c` file in Flick's ONC RPC phonebook source directory. Abbreviated code for `add_1` is shown below. Note that as expected of ONC RPC work functions, the return value is a pointer to a statically allocated result.

```
int *add_1(entry *arg, struct svc_req *obj)
{
    static int result;
    int i;

    /* Return result: zero for success, or non-zero for error. */
    result = 1;

    /* See if this entry is already in the phonebook. */
    for (i = 0; i < pb_size; ++i) {
        if (pb[i] && !strcmp(pb[i]->n, arg->n)) {
            /* We found a duplicate!  Return an error code. */
            return &result;
        }
    }

    /* Find an empty entry in 'pb'; grow the phonebook if necessary. */
    i = find_empty_entry();
    if (i == NULL_PB_INDEX)
        return &result;

    /*
     * Allocate memory for the new entry.  Note that we have to copy the
     * 'arg' data because ONC RPC says we can't keep pointers into 'in'
     * data after this function has returned.
     */
    pb[i] = (entry *) malloc(sizeof(entry));
    if (!pb[i])
        return &result;

    pb[i]->n = (char *) malloc(sizeof(char) * (strlen(arg->n) + 1));
    pb[i]->p = (char *) malloc(sizeof(char) * (strlen(arg->p) + 1));
    if (!(pb[i]->n) || !(pb[i]->p)) {
        /* Free what we have allocated and signal an exception. */
        ...
```

```
    }

    /* Copy the ‘arg’ information into our phonebook. */
    strcpy(pb[i]->n, arg->n);
    strcpy(pb[i]->p, arg->p);

    /* Increment the number of entries in our phonebook. */
    pb_elems++;

    /* Success! */
    result = 0;
    return &result;
}
```

5.3.3 Client Program

You must provide the `main` function for the client, which will allow the user to invoke the Flick-generated stubs to communicate with the ONC RPC phonebook server. The host name of the server is provided on the command line; the values of `netphone` and `firstphone` are defined in the file `phone-client.h`, which is created by Flick. The `FLICK_SERVER_LOCATION` structure is described in Section 4.3.2.

```
int main(int argc, char **argv)
{
    CLIENT client_struct, *c;
    FLICK_SERVER_LOCATION s;
    int sel, done;

    c = &client_struct;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <host>\n", argv[0]);
        exit(1);
    }

    s.server_name = argv[1];
    s.prog_num = netphone;
    s.vers_num = firstphone;
    create_client(c, s);

    done = 0;
    while (!done) {
        read_integer(("\n(1) Add an entry (2) Remove an entry "
                      "(3) Find a phone number (4) Exit: "),
                     &sel);
        switch(sel) {
        case 1:  add_entry(c); break;
        case 2:  remove_entry(c); break;
        case 3:  find_entry(c); break;
```

```
            case 4:  done = 1; break;
            default: printf("Please enter 1, 2, 3, or 4.\n");
            }
        }
        return 0;
}
```

The client's `add_entry` function invokes `add_1`, which is the Flick-generated stub for the `add` operation. Exceptions are detected by examining the return value of the stub function. A null pointer indicates an RPC failure; otherwise, `res` points to the return code for the operation.

```
void add_entry(CLIENT *c)
{
        entry e;
        char name_array[NAME_SIZE], phone_array[PHONE_SIZE];
        int *result;

        e.n = name_array;
        e.p = phone_array;

        read_string("Enter the name: ", e.n, NAME_SIZE);
        read_string("Enter the phone number: ", e.p, PHONE_SIZE);

        result = add_1(&e, c);
        if (!result)
            printf("Error: bad RPC call for add_1.\n");
        else if (*result)
            printf("Error: `%s' not added, error code = %d.\n",
                   e.n, *result);
        else
            printf("`%s' has been added.\n", e.n);
}
```

5.3.4 Compiling the Application

The `test/examples/phone/oncrpc` directory contains a simple `Makefile` for compiling the phonebook server and client programs. You will need to edit the `Makefile` slightly in order to suit your build environment. Once that is done, and you have built Flick and the ONC/TCP runtime, you should be able to type `make` to build the ONC RPC phonebook. Two programs will be created: `phoneserver`, the server, and `phonebook`, the client.

5.3.5 Using the Phonebook

To run the application you must first start the phonebook server. Note that unlike the CORBA phonebook server, our ONC RPC server does not require any command line arguments. Once the `phoneserver` program is running, you can invoke the `phonebook` program, giving it the name of the host on which the server is running.

```
1 marker:~> phoneserver
Server sendbuf: 0 2 65536
Server recvbuf: 0 2 65536

# Run a client on the same machine or on a different machine.

3 fast:~> phonebook marker
Client sendbuf: 0 0 65536
Client recvbuf: 0 0 65536

(1) Add an entry (2) Remove an entry (3) Find a phone number (4) Exit:
...
```