The ISIS System Manual, Version 2.0

K. Birman, R. Cooper, T. Joseph, K. Marzullo, M. Makpangou
K. Kane, F. Schmuck and M. Wood

September 11, 1990

Contents

Preface

This manual describes the structure and use of the ISIS toolkit for distributed and fault-tolerant programming. The material is structured so that a user unfamiliar with ISIS should be able to write and test a distributed application after reading just the first two chapters. Subsequent chapters cover the various tools provided by ISIS in much greater detail, and the reader who works through them should emerge with a thorough practical understanding of what ISIS does and how to make use of it.

Throughout the manual, an attempt has been made to include code samples as often as possible. Our hope is that in many cases, the reader will be able to copy these code samples, making only a small number of changes to adapt them into the environments where they will actually be used.

The structure of the manual is as follows:

Chapter 1. Getting Started An introduction to ISIS, focusing on an example of a small but typical ISIS application.

Chapter 2. ISIS in the Large How to approach the design of a typical ISIS application.

Chapter 3. The major components of the ISIS system A brief introduction to the structure of ISIS itself.

Chapter 4. Basic Facilities Basic data structure and ISIS system calls. How to maintain replicated data using ISIS.

Chapter 5. More About Messages A more detailed discussion of the ISIS message subsystem.

Chapter 6. The Lightweight Task Subsystem A more detailed discussion of the ISIS lightweight task subsystem.

Chapter 7. Broadcast Interface The long form of the broadcast system call, and the options it supports.

Chapter 8. Virtual Synchrony A more detailed presentation of the idea behind virtual synchrony and its impact on programming in ISIS.

Chapter 9. Replicated data A discussion of how to maintain replicated data using ISIS.

Chapter 10. Distributed and Parallel Executions Techniques for obtaining distributed and parallel executions in ISIS.

Chapter 11. Synchronization How to obtain synchronization and locking using ISIS.

Chapter 12. Transactions Dealing with transactional databases and files from within ISIS.

Chapter 13. Bypass communication Details of the new mechanisms for fast communication in ISIS.

Chapter 14. The Logging Facility How to use the ISIS logging facility to develop software that can recover from total failures without losing its state.

Chapter 15. Broadcast Types and order Types of broadcasts available and how their delivery ordering guarantees vary.

Chapter 16. Advanced Facilities Discussions of a number of advanced topics, including reception and generation of signals, forking from within ISIS clients, the remote execution facility, rules for interacting with devices from within ISIS, using ISIS in an X-windows program, using ISIS in a suntools program, the recovery manager, an interactive ISIS control program, creating and interpreting client and protocol process dumps, how the system behaves when overloaded, and adding new transport protocols to ISIS.

APPENDICIES

Appendix A. Setting Up ISIS A section aimed at the systems administrator responsible for setting up ISIS on a network.

Appendix B. Quick Reference A summary of the ISIS system.

Appendix C. Performance of the toolkit facilities Will contain detailed performance information about the broadcast primitives and the toolkit as a whole in a future version of the manual.

Appendix D. Demonstration programs ISIS comes with several demo programs. This appendix explains how to run them.

Appendix E. FORTRAN interface to ISIS ISIS can be called from UNIX F77. This appendix summarizes the changes to the ISIS interface made to support such calls. Currently, the F77 interface has only been tested under SUN OS, but it should work under other systems such as MACH as well.

Appendix F. Dealing with old code and non-ISIS task packages Some systems on which ISIS runs (MACH, SUN OS 4.0, APOLLO UNIX) support their own lightweight task/process mechanisms. You can use these from ISIS, preemptive scheduling and all. This chapter explains how. The chapter also covers some issues that arise when integrating pre-existing programs into an ISIS-based application.

Appendix G. Using ISIS from LISP ISIS can be called from Allegro Common LISP and LUCID Common LISP (we are working on Harlequin). This chapter covers the necessary details.

Appendix H. The META System META is a system for defining and monitoring realtime sensors under ISIS and for triggering actions based on detected events.

Changes to ISIS in switching from V1.3.1 to V2.1

The purpose of this section is to summarize the ways that ISIS has changed in going from ISIS V1.3.1 to V2.1. Most changes are upgrades that continue to support existing code without requiring modifications.

1. This manual has been extensively revised and has major new sections on issues such as large-systems architecture and long-haul communication. Research papers are available on most of the major changes, through Cornell.

2. The new BYPASS facility is working quite well even for many groups and rapid group membership changes. You need to enable this at compile time and you need to be consistent in any given group (either all members link with a "bypass" copy of clib or none do so). In ISIS V3.0 BYPASS will be the default. Right now, because `pg_client` doesn't exploit the BYPASS protocols we don't enable it by default, but this restriction will soon be eliminated.

3. The META system has been substantially extended and is described in Appendix H.

4. There are new ways to connect to ISIS. The interface `isis_init_l` offers a way to force a restart of the system if an application starts up and ISIS is not running; it also gives some control over whether ISIS will "panic" if the system is not up. Restart is done, if necessary, by running the shell script `/usr/bin/startisis`. *You must define and install this shell script if you plan to make use of this feature.*

5. There is a new interface, `isis_remote`, by which remote clients (on machines not listed in the ISIS sites file) can connect to the system at some "mother" location, obtaining all of its features transparently. Initially this will support only remote UNIX clients, but we hope to extend the mechanism to support remote clients on other host operating systems in the future (OS/2, DEC VMS, and perhaps even IBM's VM system). In the present version of the system, if the mother machine for a remote client fails, the remote client can trap the resulting exception by defining a procedure `isis_failed()` that reconnects to ISIS on a different machine and returns 0; it will, however, be necessary to rejoin any process groups to which the application belonged each time this occurs. The whole mechanism will be made more transparent in ISIS V3.0.

6. Both `isis_init` and `isis_remote` now check for environment variables that might define the ISIS port number to connect to. `isis_init` checks for the variable `ISISPORT` and uses it if found. `isis_remote` checks for the variable `ISISREMOTE` and uses it if found.

7. Associated with this interface is a new routine `isis_probe(freq,timeout)` that asks ISIS to check the liveness of a client at frequence `freq` seconds, shutting it down if there is no reply in `timeout` seconds.

8. A set of new mechanisms have been added for faster communication. These are called the *bypass* protocol suite and include a group subset communication option (called *process lists*) and a way to define user-supplied data transport routines. A minimal interface to the transport layer has also been added, called `mbcast`.

9. The address structure has been changed, and the routine `addr_cmp` no longer looks at entry numbers at all. The old semantics of `addr_cmp`, in which the entry numbers are compared under a wild-card rule, are still available through an interface called `paddr_cmp`.

10. The problem of group address pointers being deallocated when a process leaves the group has been eliminated. Group addresses are now cached and the pointers remain valid indefinitely. The overhead of this is low. The approach also makes `pg_lookup` much cheaper.

11. A new spooling and long-haul communication facility has been added to the system. It can be used to build services that only run periodically, and to interconnect physically remote ISIS clusters.

12. The number of sites that can be connected to ISIS has been greatly increased, to 255 per cluster plus an unlimited number of machines using the `isis_remote` interface. The `sites` file only lists the sites directly in the cluster.

13. The limit on the number of members and clients in a process group has been *decreased* to 32 because the BYPASS protocol is slow for group membership changes with more than this number of members. If you don't use BYPASS you could set PG_ALEN in protos/pr_groups.h to a higher value (it used to be 128). We plan to eliminate the limit on clients completely in V3.0 of the system, but the limit on members is probably not going away anytime soon.

14. A new routine `cc_terminate_l` is supported, it sends a copy of the termination message to an additional group or process destination as well as terminating the coordinator-cohort computation, all in a single atomic action. When the extra address refers to a group, the caller must belong to that group.

15. A new routine `pg_detect_failure` is available for detecting total failure in a group to which the caller does not belong.

16. The runtime memory requirements of the system have been reduced.

17. ISIS can now be accessed from C++ and compiled or called from GCC. The necessary type signatures are included.

18. Some routines have become macros, such as `msg_delete` and `addr_cmp`. This may result in problems compiling code that passed such routines as addresses (there are usually "real" versions of them around too, for example `MSG_DELETE` and `ADDR_CMP`).

19. The task facility has been extended to support a task-level *select* mechanism, called `isis_wait`, as well as a way to enter and leave the ISIS tasking world dynamically, i.e. to exploit true parallelism on a multiprocessor.

20. To benefit from the Mach copy-on-write message passing mechanism, ISIS now uses Mach IPC if possible for its intra-machine communication. The change is transparent but leads to a substantial performance improvement within Mach systems.

21. Ways to refuse to participate in a state transfer or coordinator-cohort computation have been added.

22. The client dump format has been extended and improved.

23. The SUNTOOLS graphics interface is being phased out in favor of the various forms of X11, including Open Look. We are looking at improvements to the X11 interface at the widget level, which is currently not an option in ISIS The recommended X11 interface has changed a bit (see the spread or grid demos for examples).

24. A new message library has been added that includes a way to put an indirect data reference into a message, using a format `%*X`. ISIS does a call-back to a user-supplied routine when finished with the pointer. By combining this with the `%-X` format type in `msg_get`, all data copying can be eliminated.

25. Support for the floating point and double-precision data types has been added. ISIS assumes that the IEEE standard floating point format is in use.

26. The remote-exec utility has been extended to implement the remote user-id and password options, and the parallel make demo has been fixed.

27. An all-out attack on performance has greatly improved the speed of the whole system.

28. The broadcast "guard" facility has been eliminated.

29. A bug prevents the simultaneous use of SUNTOOLS graphics application and the new bypass mode software; things are fine if ISIS is compiled with BYPASS disabled (no -DBYPASS on command line when compiling `clib/cl_bypass.c`). However, compiling this way slows things down to the performance of the old V1.3.1 system.

30. The fortran interface now supports function calls with underscores in the variable names for use from fortran's that permit such names. It is illegal to mix both styles of reference, however.

31. A bug in the SUN4 lightweight context switch code was fixed, permitting larger numbers of tasks and faster task-to-task switching.

32. An unsupported feature permitting users to run multiple ISIS systems on a single machine (i.e. to debug code that senses and reacts to remote hardware failures) is now supported (see Chapter 16).

33. A new style of message reception is supported. You specify the entry routine as `MSG_ENQUEUE` and use `msg_rcv` to collect the arriving messages one by one. There are performance advantages but many virtual synchrony risks to doing this.

34. Timeouts are supported in `isis_accept_events()` and in `bcast_l`.

Looking further into the future, we expect ISIS V3.0 to be available sometime in late 1990. That version of the system will will tolerate network partitioning and support scaling mechanisms suitable for use in extremely large networks, with potentially thousands of nodes.

Information on future releases of ISIS is available from Cornell University Contact isis@cs.cornell.edu to be added to the ISIS mailing list for new technical reports or to receive printed notification of new releases. We also urge ISIS users to follow the network newsgroup "comp.sys.isis" for discussion of ISIS-related topics, bug fixes, and so forth.

ISIS bugs should be reported to isis-bugs@cs.cornell.edu. We respond promptly to any and all reports, however minor. If you have problems installing ISIS on your system, we will be happy to help.

Chapter 1

Getting Started

There is perhaps no better way to learn about a new system than by actually using it, so let us begin by using ISIS to develop a simple application. A typical ISIS application has parts that run on several different machines and can be expanded to include more machines or be removed from some machines even as the application is running. The reason for distributing the application over a number of machines may be to share the work, to obtain faster response time, or to be able to continue operation despite the failures of some of the machines. The example below has been chosen with the idea of exposing you to most of the basic features of ISIS rather than to illustrate a real-life ISIS application. Yet, you will see that ISIS makes it simple to write a program that is distributed, dynamically expandable, and fault-tolerant.

We will consider a distributed "time-card" service for an organization with several departments. The organization hires a number of temporary workers, who may work in several different departments in a given week, covering excess work where and when required. Each department separately records the number of hours each temporary employee works in that department. The object of the time-card service is to enable someone to give the service the name of an employee and obtain the number of hours that the employee worked in the various departments the previous week. (The employee will presumably be paid on this basis.) The time-card service will have to search through the records of the individual departments before giving its response.

If the organization has a number of workstations connected by a network (and they have ISIS), this service could be implemented to run on several

of these machines instead of on just one machine. One advantage is that the records of different departments can be scanned in parallel on several machines, instead of one after the other on a single machine. This means that queries will be answered sooner. A second advantage is that we can ensure that the service remains available even when some of the machines are not operational because of a failure or because they have been taken off-line for maintenance or for other reasons. Let us assume that each department keeps its records in a file called `department1`, `department2`, `department3`, and so on, and that each record is simply a line giving the employee's name followed by the number of hours he or she worked. We will also assume that these files are available on all the machines on which the service runs.

Before we go any further with our example, we need to introduce some of the ideas that are central to programming with ISIS.

Process groups and broadcasts

One of the most basic mechanisms that ISIS provides is a means of grouping processes together and naming them as a unit. A process group could contain just a single member, but will often consist of a number of processes residing on machines anywhere in the system. The membership of a process group could change with time, as new processes join the group or as existing members leave it, either out of choice or because of a failure of some part of the system. A process can be a member of more than one process group.

ISIS also provides a broadcast mechanism that enables you to send a message from a process to a process group. To do this, the sending process first asks ISIS to "look up" the name of the process group and obtains an "address". It then performs a broadcast[1] giving this address, the message, and other relevant information as arguments. The effect of this is to send a copy of the message to each of the current members of the process group. All members will eventually receive the message, although they may receive it at different times.

The broadcast mechanism also allows the recipient of a message to send a reply, or to forward it to some other process that will send a reply. A process broadcasting a message can indicate that it wants to wait for a specific number of replies, or that it wants a reply from all the recipients of the message. The broadcast function call returns when the requested

[1] The term "broadcast" is used in many different ways; we use it to mean the sending of a message from one process to others using the ISIS broadcast function call.

number of replies have been received. If the group is not large enough, or if so many recipients terminate (possibly because of failures) that the required number of replies cannot be collected, ISIS will collect as many replies as possible and notify the sender of the shortfall. This reply mechanism allows the ISIS broadcast facility to be used as a generalized remote procedure call mechanism.

The most common reason for making a set of processes into a process group is to be able to broadcast messages to the group as a whole, even when its membership may be changing. As a special case, the simplest way to send a message to an individual process is to make it a member of a process group containing only itself, and broadcast messages to this group. If a process will receive messages both as an individual and as a member of a process group, it can be made a member of two process groups. Process groups are cheap in ISIS, so this should not pose performance problems.

There is another reason why you might want to make a set of processes into a process group. ISIS provides simple-to-use tools that permit members of a process group to access shared or replicated information, to perform certain forms of coordinated distributed execution, and to tolerate and recover from failures, among other things. If a set of processes wishes to make use of these tools, they would typically be made members of a process group. These tools will be described in later sections.

Process groups provide a convenient way of giving an abstract name to the service implemented by the members of the group. Other processes interact with the service using the name of the group and the broadcast facility. They need not be aware of the actual membership of the group. This means that the group implementing the service can grow, shrink, move to different machines, or add new capabilities without any interruption to the service, and with the users of the service being unaware of these changes. It is this feature that makes it possible to develop applications that are modular, dynamically expandable, and tolerant of failures. Figure 1.1 provides an illustration of this. It shows a process P communicating with a process group that implements a print service. The important thing to notice is that in all three cases P addressed its message to the group `PrintService`. ISIS keeps track of the group membership and delivers the message to the current members. P thus thinks of the group as an abstract service implementing some function (print documents) and does not care about how many members the group has or where they happen to be currently located.

It should be noted that the ISIS namespace is actually structured into naming "scopes" in order to limit the cost associated with this addressing

Figure 1.1: Process groups provide a unit of abstraction.

mechanism. However, one can use ISIS without worrying about this issue, and we will defer discussion of it until later in the manual.

Tasks and entries

Readers familiar with UNIX will know that a UNIX "process" has a private address space within which a single thread of control lives. Some newer operating systems like MACH have introduced a notion of lightweight tasks that coexist within a single process, sharing the address space. Although ISIS was built on top of UNIX, we needed a task mechanism to implement the system. Consequently, a process in an ISIS application is internally structured into a number of tasks. An ISIS task looks just like a C function, and shares the same address space and global variables as all the other tasks and functions in the process. The difference is that a task (and not just the function called `main`) can be invoked by the system and start executing in response to certain events, the most common of which is message delivery. A task that is started up in response to a message delivery is called an "entry". A process can have many entries and each one is given a different "entry number". When a message is sent, it is addressed to a particular entry number. On delivery, (a pointer to) the message is passed as a parameter to the entry, which typically reads the contents of the message and acts accordingly.

Programming with tasks is not very different from programming with regular C functions, except for three things. One is that you may need to link to special libraries, such as "-llwp" under SUN UNIX. Another is that when a task makes certain ISIS system calls, it is possible for a new task to be started up and begin executing before the system call returns. The original task will later continue from where it left off. As an example, consider a task that made a system call to broadcast a message and is now waiting for replies. If a new message arrives before the replies come, another task (an entry) may be started up to handle the new message even though the first task has not terminated. Normally this poses no problem, but if the second task changes the values of global variables, then the first task has to be aware of the fact that their values might change between when it performs the broadcast system call and when the system call returns. System calls that allow other tasks to be started up before they return are called "blocking" system calls because they "block" the execution of the task that performs the call. This documentation will indicate which system

calls may block, and under what conditions.

The other thing to keep in mind when programming with tasks is that they may have a stack limit of 32k bytes. We say "may" because this limit does not apply on all systems (c.f. MACH) and because it can be over-ridden if necessary. A 32kb stack is sufficient for most purposes, but extremely deep nesting of function calls (as might happen with recursive calls) or allocation of large arrays or data structures on the stack may lead to the limit being exceeded. ISIS will usually, but not always, detect this. You can increase (or decrease) this limit with a one-time declaration, but be aware that *all* tasks will have this new stack size, and if it is made too large you may have problems with memory allocation. There are various ways to get around this, including setting a per-task limit or calling a single subroutine without any stack size limit, provided that the subroutine will not invoke any ISIS functions before it returns. This is convenient when, for example, an ISIS task must call a large piece of software over which the ISIS programmer has no control, and which might not respect the ISIS conventions. We'll say more about this later.

Considerable thought has been given to the problem of porting "old code" to run under ISIS, especially in the case where the old code was not task-oriented. By taking appropriate care, one can port "old" programs to ISIS in such a way that only new code (added in conjunction with the port) is subject to any stack limit at all. Moreover, under systems that already support a lightweight task facility, such as SUNOS (LWP) or MACH (Cthreads), ISIS allows you to combine the "native" facilities with the ISIS facilities in a completely unrestricted manner. See Appendix F for a detailed discussion of both of these subjects.

Monitoring events

A process can instruct ISIS to notify it when certain types of events occur. It does so by giving ISIS the name of a task to invoke (and an argument to pass to the task) when that type of event takes place. Among the types of events that can be monitored are process group membership changes, process termination, and site failures. This facility may be used to reapportion the work load when new members join or leave a process group, or to take over work from a process or site that fails, among other things.

Example: the time-card service

From the discussion above, it follows that the time-card service should be implemented as a process group consisting of a number of processes running on different machines. We will call the group `timeservice`. The reason for putting the processes in one process group is to be able to use the broadcast mechanism to send queries to the group as a whole. All the members of the time-card service execute the same program (which we will call the service program). Another program (the query program) is used to query the service. We will develop both programs side by side, as is typical when programming with ISIS.

Part of the code for the service program is shown below. The value `<port-no>` is the port number used by ISIS to talk to applications. You will have to ask your system administrator for this. If the port-number is given as 0, then ISIS will first check for an environment variable ISISPORT, and will use this number if found. If not, ISIS will look in the `/etc/services` file, you may give the value 0 for `<port-no>` and ISIS will look it up for you. You may need to consult with the person who installed ISIS on your system if you try using port number 0 and your program won't start up. ISIS V2.0 and beyond includes automated restart procedures that start ISIS on a site when the first attempt is made to run an ISIS application there. This is done using the interface `isis_init_1`, as described in Sec. 2.9. To connect to an ISIS running on a different machine, use `isis_remote`, as described in Sec. 2.10. A call to `isis_probe(freq,wait)` is used to tell ISIS to begin watching th e client. ISIS will probe it once every `freq` seconds and kill the process is no response if received after `wait` seconds. By default, ISIS will not probe local clients and will probe remote clients every 60 seconds, killing them if there is no response within a further 60 seconds.

```
#include    "isis.h"
#define     QUERY_ENTRY     1

address     *gaddr_p;
int         my_index;
int         my_dept;

main()
{
    int     service_maintask();
```

```
    int     group_change();
    int     receive_query();

    isis_init (<port-no>);

    /* Declare tasks and entry points */
    isis_task (service_maintask, "service_maintask");
    isis_task (group_change, "group_change");
    isis_entry (QUERY_ENTRY, receive_query,
                                    "receive_query");

    isis_mainloop (service_maintask);
}

service_maintask()
{
    int     group_change();

    /* Join the process group and monitor */
    /* membership changes                 */
    gaddr_p = pg_join ("timeservice",
                        PG_MONITOR, group_change, 0,
                  0);
    isis_start_done();
}

group_change (gview_p, arg)
  groupview *gview_p;
  int       arg;
{
    int     i;

    /* Compute a unique index for this member */
    i = 0;
    while (!addr_ismine (&gview_p->gv_members[i]))
        i++;
    my_index = i + 1;
}
```

In an ISIS application, the function `main` usually just reads in the command line arguments (this example has none), initializes ISIS, declares tasks and entries, and sets off the main loop. The argument to `isis_mainloop` is the first task to be run. The first thing that `service_maintask` does is to join the process group `timeservice` and set up a monitor for group membership changes. The first argument to `pg_join` is the name of the group, and the last argument is always a 0. In between these two arguments, you may specify a number of optional keywords and the arguments corresponding to those keywords. Here, the keyword `PG_MONITOR` specifies that the task `group_change` is to be called with the new group view and the given argument (in this case 0) whenever the group membership changes. The function `pg_join` returns the address of the group, which is stored in the global variable `gaddr`. When the main task is begun, ISIS inhibits the delivery of new requests from other processes. This ensures that you can do all the necessary initialization before being asked to respond to other events like incoming messages. The call `isis_start_done` tells the ISIS system that the startup sequence is completed. This means that new tasks may be started up at the next blocking system call or after the main task terminates. ISIS automatically invokes `isis_start_done` when the main task terminates (the call is hence unnecessary here), but if the main task remains in a loop and you forget to call `isis_start_done`, your application will simply execute the main task and do nothing else. Notice that if you want to terminate the execution of an ISIS process, you must call `exit` explicitly.

It is important to realize that an ISIS process can be active even if there are no active tasks within it (e.g. the main task has finished, and no new tasks have been started). In fact, ISIS is designed under the assumption that tasks will start up, do the work they are supposed to do, and then terminate (by returning). This applies to the main task as well as any others. A process with no active tasks in it is in fact waiting in `isis_mainloop` for work to do. Later in the manual we describe a way to obtain a printable dump of the internal state of a process that includes a list of all active tasks within it.

Look now at the routine `group_change`. This routine is called in each member of the group (recall that they are all executing the same piece of code) whenever the group membership changes. Routines that monitor group memberships are always called with a pointer to the "group view" structure as its first argument. (The second argument is a value supplied by the user when the monitor is set up). The `groupview` structure contains information about the process group (Section 2.5). In particular, it contains

a list of the addresses of all the current members `gv_members`. The group view structure always orders this list according to the "rank" of the members. The oldest member in the group has rank 0, the second oldest rank 1, and so on. So the rank can be used to give each member a unique index that distinguishes it from the other members. In the example, `my_index` contains the value of the rank +1. (Another way to obtain the rank is to call `pg_rank(gaddr, paddr)`, which returns the rank of process `paddr` in the group `gaddr`.) The task `group_change` is invoked in a process when the process joins the group for the first time (this, too, is a membership change), so when `isis_start_done` is called `my_index` will have a defined value. This index will change each time a member joins or leaves the group.

If you actually type this example in, you may wonder how to print things like ISIS address data structures in a human-readible format. ISIS has functions for this purpose. For example, `paddr` will print a single address (given a pointer to it), and `paddrs` will print all the addresses in a list of addresses, such as the one `msg_getdests` (get destinations to which a message was sent) returns. The fields printed include the site-id and incarnation number where the process is running and the UNIX process-id of the process. For example, to print the the contents of a groupview *gip*:

```
print("Group <%s>: ", gip->gi_name);
paddr(gip->gi_addr);
print(" %d members, viewid %d.%d\n",
    gip->gi_nmemb, VMM(gip->gi_viewid));
print("Members = [");
paddrs(gip->gi_members);
print("]\n");
```

Here, the macro `VMM` is used to take the unsigned long integer `viewid` apart into its *major* and *minor* numbers. Major numbers change only when processes join and leave a group; the minor viewid number also increments when certain types of (very infrequent) broadcasts are received by the group members.

To receive incoming messages we must define an entry. The `isis_entry` statement declares such an entry, giving it the entry number `QUERY_ENTRY`. We now give the code for this entry. We assume that an incoming query message contains a string giving the name of an employee. For now, we assume that there are at least as many members in the process group as there are departments, and that each member is responsible for searching through the file for the department whose number is in `my_dept`. Extra

members have the value 0 in `my_dept` and do nothing. We will see later how the members decide which department they are responsible for, and how they handle the case where there are fewer members than departments.

```
receive_query (msg_p)
  message   *msg_p;
{
    char    query_name[MAX_NAMELEN];
    int     query_hours;

    if (my_dept != 0)
    {
        /* Read employee name from message */
        msg_get (msg_p, "%s", query_name);

        /* Search through relevant file to find number */
        /* of hours worked in my_dept and store it in  */
        /* query_hours                                 */

        /* Send reply message */
        reply (msg_p, "%d%d", my_dept, query_hours);
    }
    else    /* I am not responsible for any department */
        reply (msg_p, "%d%d", 0, 0);
}
```

An entry is called when a message addressed to its entry number arrives at a process. Its first argument is a pointer to the message. This pointer may be used to read data out of the message using `msg_get`, which has an interface similar to `fscanf`. In this case, it reads a string of characters out of the message and stores it in `query_name`.

Let us shift gears for a minute and look at the query program. It has a similar initialization sequence, and simply reads in a name from the terminal, broadcasts a message to the `timeservice` process group, and prints out the replies. This is what the code looks like.

```
#include    "isis.h"
#define     QUERY_ENTRY     1           /* From the service */
#define     MAX_NAMELEN     64
```

```
#define     NDEPTS          5

main()
{
    int     query_maintask();

    isis_init (<port-no>);

    /* Declare tasks and entry points */
    isis_task (query_maintask, "query_maintask");

    isis_mainloop (query_maintask);
}

query_maintask()
{
    address *gaddr_p;
    char    name[MAX_NAMELEN];
    int     dept[NDEPTS], hours[NDEPTS];
    int     i;

    /* Find address of timeservice process group */
    gaddr_p = pg_lookup ("timeservice");
    if (addr_isnull(gaddr_p))
    {
        printf ("Sorry! the service is not available\n");
        exit();
    }
    isis_start_done();

    /* Loop asking queries */
    printf ("Enter employee name (^D to quit): ");
    while (scanf ("%s", name) == 1)
    {
        /* Broadcast a message containing the name and */
        /* collect replies                             */
        do
            bcast (gaddr_p, QUERY_ENTRY, "%s", name, ALL,
                                    "%d%d", dept, hours);
```

```
        while (isis_nreplies != isis_nsent);

        /* Print out time card */
        printf ("Time card for %s:\n", name);
        printf ("    Dept.   Hours\n");
        for (i = 0; i < isis_nreplies; i++)
            if(dept[i])
                printf ("%8d%8d\n", dept[i], hours[i]);

        /* Read in next name */
        printf ("\nEnter employee name (^D to quit): ");
    }

    /* Quit by explicitly terminating this process */
    exit(0);
}
```

Notice the use of `pg_lookup` to obtain the address of a process group. The most significant part of the code, of course, is the call to `bcast` to send a message and collect the replies. As you can see, the call first specifies the address and the entry number. This is followed by a description of the data to be put into the outgoing message (in a form similar to `fprintf`). Next comes the number of replies wanted. The constant `ALL` specifies that a reply is wanted from all the processes to which a copy of this message was sent. This is followed by a description of where to put the data that is read out of the reply messages (in a form similar to `fscanf`). Unlike `fscanf`, though, each item is a pointer to an *array* of the given type, because there could be more than one reply message. The data from each reply goes into one element of each of the arrays. Compare the call to `bcast` in the query program with the calls to `msg_get` and `reply` in the service program; it's easy to see how they match up.

When a call to `bcast` returns, the global variable `isis_nsent` contains the number of processes that were sent a copy of the message and `isis_nreplies` contains the number of replies collected (not counting "null replies" sent using the special ISIS function `nullreply`). (`isis_nreplies` is also the return value of `bcast`, unless an error occurs, in which case an error code is returned.) In our example, these two values would normally be equal. However, if a process that was sent a message terminates before replying (possibly because of a failure), ISIS will detect this and the call

to `bcast` will return without collecting a reply from this member. A process that terminates is automatically dropped out of any process group it belonged to, so our query program reissues the broadcast if this happens. This next time around the message will be sent to the new membership of the group and unless more processes terminate, a reply will be received from all of them.

Now we shall see how each member is assigned a department (we still assume that there are at least as many members as there are departments). Let `NDEPTS` be the number of departments. Earlier we showed how each member can compute a unique index (`my_index`). A simple rule would be to make the member with index i responsible for department i, for $i \leq$ `NDEPTS`. A member with index $i >$ `NDEPTS` does nothing unless an active member drops out of the group (possibly because of a failure). If the index of a previously inactive member now becomes less than or equal to `NDEPTS`, it will begin to take part in the search. Such a process is called a "standby". Standbys make an application tolerant of failures, and are used often in ISIS. This application will tolerate the failure of as many processes as there are standbys.

The complete code for the service program is given below. Note that whenever the group membership changes, each member recomputes `my_index` and opens the relevant file `department`i (and closes any previously opened one).

```
#include    <stdio.h>
#include    "isis.h"
#define     QUERY_ENTRY     1
#define     MAX_NAMELEN     64
#define     NDEPTS          5

address     *gaddr_p;
int         my_index;
int         my_dept = 0;
FILE        *my_file_p;

main()
{
    /* Same as before */
}
```

```
service_maintask()
{
    /* Same as before */
}

group_change (gview_p, arg)
  groupview *gview_p;
  int       arg;
{
    char    filename[16];

    /* Compute a unique index for this member */
    i = 0;
    while (!addr_ismine (&gview_p->gv_members[i]))
        i++;
    my_index = i + 1;

    /* Close previously open file, if any */
    if (my_dept != 0)
        fclose (my_file_p);

    /* Reassign departments */
    if (my_index <= NDEPTS)
    {
        my_dept = my_index;

        /* Open relevant file */
        sprintf (filename, "department%d", my_dept);
        my_file_p = fopen (filename, "r");
        if (my_file_p == NULL)
        {
            printf ("Could not open file %s\n", filename);
            exit();
        }
    }
    else
        my_dept = 0;
}
```

```
receive_query (msg_p)
  message   *msg_p;
{
    char    query_name[MAX_NAMELEN], name[MAX_NAMELEN];
    int     query_hours, hours;

    if (my_dept != 0)
    {
        /* Read employee name from message */
        msg_get (msg_p, "%s", query_name);

        /* Search through relevant file to find number */
        /* of hours worked in my_dept                  */
        query_hours = 0;
        while (fscanf (my_file_p, "%s %d", name,
                                          &hours) == 2)
            if (strcmp (query_name, name) == 0)
            {
                query_hours = hours;
                break;
            }

        /* Send reply message */
        reply (msg_p, "%d%d", my_dept, query_hours);
    }
    else    /* I am not responsible for any department */
        reply (msg_p, "%d%d"", 0, 0);  /* Say I won't be replying */
}
```

We now have a complete implementation for the time-card service. At this point you can compile the two programs separately and link each of them with the ISIS libraries `isislib1.a`, `isislib2.a`, and `isislibm.a` (in that order). (The code and some sample data files are provided in the `demo` directory. Check with your site administrator to find out where these subroutine libraries are located on your machine.) You can then start up as many instances of the service program as you wish, perhaps on different machines, and as many instances of the query program as you want, also on different machines if desired, and begin issuing queries. To test the fault tolerance of the program, you may wish to add a loop to the query program,

so that for each name you type in, it broadcasts the same query over and over, say 50 times. Then as the service is being queried repeatedly, you can add new members by starting up new instances of the service program on any machine or simulate a failure by killing existing members (using `^C` or the `kill` command). You will see that as long as there are at least `NDEPTS` members, the sgrvice will continue to operate correctly. You can even turn off the power from a machine that has a member on it. In this case, there may be a pause because ISIS will wait for an answer from this machine for about 45 seconds before timing out and deciding that it has failed (NB: this delay varies with the setting of the "-f" parameter to the ISIS protos server; see Appendix A for details). To make this even more interesting and fun, we provide the code for an implementation that works even if there are fewer than `NDEPTS` members. The changes are simple. If the number of members drops below `NDEPTS`, some members take care of more than one department. The variables `my_dept` and `my_file_p` become arrays, and the replies carry arrays, too. Notice the changes to `bcast`, `msg_get`, and `reply` to handle arrays. The details of the syntax are given in Section 4.3. Here is the service program.

If you kill a machine, there may be a long delay before ISIS notices, depending on how it was configured at your site.

```
#include    <stdio.h>
#include    "isis.h"
#define     QUERY_ENTRY     1
#define     MAX_NAMELEN     64
#define     NDEPTS          5

address     *gaddr_p;
int         my_index;
int         my_ndepts = 0;
int         my_dept[NDEPTS];
FILE        *my_file_p[NDEPTS];

main()
{
    /* Same as before */
}

service_maintask()
{
    /* Same as before */
```

```
}

group_change (gview_p, arg)
  groupview *gview_p;
  int       arg;
{
    char    filename[16];
    int     n_members;
    int     i;

    /* Compute a unique index for this member */
    i = 0;
    while (!addr_ismine (&gview_p->gv_members[i]))
        i++;
    my_index = i + 1;

    /* Record number of members in group */
    n_members = gview_p->gv_nmemb;

    /* Close previously open files, if any */
    for (i = 0; i < my_ndepts; i++)
        fclose (my_file_p[i]);

    /* Reassign departments */
    my_ndepts = 0;
    for (i = my_index; i <= NDEPTS; i += n_members)
    {
        my_dept[my_ndepts] = i;

        /* Open relevant file */
        sprintf (filename, "department%d", i);
        my_file_p[my_ndepts] = fopen (filename, "r");
        if (my_file_p[my_ndepts] == NULL)
        {
            printf ("Could not open file %s\n", filename);
            exit();
        }
        my_ndepts++;
    }
```

```
}

receive_query (msg_p)
  message   *msg_p;
{
    char    query_name[MAX_NAMELEN], name[MAX_NAMELEN];
    int     query_hours[NDEPTS], hours;
    int     i;

    if (my_ndepts > 0)
    {
        /* Read employee name from message */
        msg_get (msg_p, "%s", query_name);

        /* Search through relevant files to find number */
        /* of hours worked in my_dept[i] and store in   */
        /* query_hours[i]                               */
        for (i = 0; i < my_ndepts; i++)
        {
            fseek(my_file[i], 0, 0);
            query_hours[i] = 0;
            while (fscanf (my_file_p[i], "%s %d", name,
                                          &hours) == 2)
            if (strcmp (query_name, name) == 0)
            {
                query_hours[i] = hours;
                break;
            }
        }

        /* Send reply message */
        reply (msg_p, "%D%D", my_dept, my_ndepts,
                                  query_hours, my_ndepts);
    }
    else    /* I am not responsible for any department */
        reply (msg_p, "%d%d"", 0, 0);  /* Say I won't be replying */
}
```

And here is the corresponding query program.

```
#include    "isis.h"
#define     QUERY_ENTRY     1
#define     MAX_NAMELEN     64
#define     NDEPTS          5

main()
{
    /* Same as before */
}

query_maintask()
{
    address *gaddr_p;
    char    name[MAX_NAMELEN];
    int     dept[NDEPTS * NDEPTS], hours[NDEPTS * NDEPTS];
    int     i, j, k, rval;
    int     arraylen_1[NDEPTS], arraylen_2[NDEPTS];

    /* Find address of timeservice process group */
    gaddr_p = pg_lookup ("timeservice");
    if (addr_isnull (gaddr_p))
    {
        printf ("Sorry! the service is not available\n");
        exit();
    }
    isis_start_done();

    /* Loop asking queries */
    printf ("Enter employee name (^D to quit): ");
    while (scanf ("%s", name) == 1)
    {
        /* Broadcast a message containing the name and collect */
        /* replies                                             */
        do
            rval = bcast (gaddr_p, QUERY_ENTRY, "%s", name,
                                ALL, "%D%D", dept, arraylen_1,
                                             hours, arraylen_2);
        while (isis_nreplies != isis_nsent);
```

```
        /* Exit on error */
        if (rval <= 0)
        {
            isis_perror("Sorry! bcast error");
            exit();
        }

        /* Print out time card */
        printf ("Time card for %s (based on %d replies):\n",
                                                name, nreplies);
        printf ("    Dept.   Hours\n");
        for (i = 0, k = 0; i < isis_nreplies; i++)
            if(dept[i] == 0)
continue;
            else for (j = 0; j < arraylen_1[i]; j++)
            {
                printf ("%8d%8d\n", dept[k], hours[k]);
                k++;
            }

        /* Read in next name */
        printf ("\nEnter employee name (^D to quit): ");
    }
}
```

Exercise

As an exercise, you may wish to add an "update" entry to the service program and write a corresponding update program. The update program should read in new data for each department from the terminal, and send this data in a message addressed to the update entry number of the service. The update entry should read the data out of the message and rewrite the files `department1`, `department2`, ... If you have a shared file system, each member should rewrite only the files corresponding to the departments it is responsible for; if you have separate copies of the files at each member, each member should update all the files. An interesting observation is that because ISIS ensures that message delivery events occur in the same order everywhere, you can continue to send query messages to the service even as

an update message is being sent. It will never be the case that some members respond to a query based on old information, while others respond based on updated information—an important property for many applications.

But why does it work?

It is possible that you don't believe that the program above will work. Here's what seems to be a counter-example. Consider a query that is being sent to the service just about the same time as one of its members fails. Assume that the broadcast message reaches some members before the failure is noticed there (i.e. before the routine `group_change` is called there), while it reaches the other members after `group_change` is called. The first set of members respond based on the departments assigned to them before the failure, while the second set respond based on the new assignment of departments. Clearly, it is now possible for two reply messages to contain information about the same department, while the files for some other departments may not be searched at all. One of the main features of ISIS is that *such anomalous orderings do not happen.* The program above works correctly only because the notification of membership changes and the delivery of broadcast messages occur in the same order at all the members. ISIS guarantees that all events (including broadcast message deliveries, monitor notification, and a host of other functions to be discussed later) occur in the same order at all processes. This is true even for the notification of unpredictable events like process or site failures. It is this feature that makes programming with ISIS so simple. In the absence of this ordering guarantee, every query in our example would have to involve some kind of agreement protocol to ensure that the members agreed on the current state of the group. This not only muddles the application, but also leads to rather poor performance. ISIS insulates programmers from this level of complexity. This allows them to work in a simple and ordered environment and makes it possible to build distributed applications that would otherwise be intractable. We expand upon this concept below.

Ordering in ISIS

To illustrate how the ordering of events works in ISIS, let us first look at what an execution might look like if we didn't have any ordering guarantees. Figure 1.2 shows three processes A, B, and C as they join and leave the

Figure 1.2: An execution in a disordered world.

process group `Pgroup`, while two other processes P and Q send messages to this group. The figure also shows (in curly braces) the view each process has of the current group membership. Notice the confusion. A receives $m1$ before $m2$, while B receives $m2$ before $m1$. If A and B were maintaining copies of the same data structure, for example, and $m1$ and $m2$ were requests to perform operations on this data structure (e.g. add an item to a queue, or remove the first item on the queue), then performing them in different orders could lead to the copies of the data structure becoming inconsistent. So additional communication is necessary before A or B can act on any incoming request to avoid such inconsistencies. Further in the execution we see that A receives $m3$ when it thinks that `Pgroup` consists of just A and B, while B and C receive it when they think the group contains A, B and C. We have already seen in our time service example that handling a message based on inconsistent views of the group membership could lead to incorrect results. Again, the only way to avoid this is to run some kind of agreement protocol for every incoming request. The figure also shows $m4$ being delivered to processes with inconsistent group views, this time because of a failure.

How does ISIS help? A programmer would like to think of actions like the delivery of a broadcast message or the notification of a group membership change as a single event, even though they consist of parts that take place in more than one process and possibly at different times. For example, one would like to be able to write a program thinking, "When the group receives the message broadcast by process A, do something," or "when the group is notified of the failure of process B, do something else." One look at Figure 1.2 should convince you that this kind of thinking is not possible in a disordered environment. The programmer is forced to consider the possible interleavings of the various events and cannot think of distributed events like message delivery or failure detection as single units. A programmer using ISIS, on the other hand, is guaranteed that distributed events like broadcast message deliveries, notifications of group membership changes (even if they are due to failures), and many other kinds of events will occur in exactly the same order in every process. In other words, interleavings like those in Figure 1.2 will simply never happen in ISIS. Figure 1.3 shows an execution that *could* occur in ISIS with the same set of events. Notice how much simpler things become. A programmer can work with the knowledge that each process has the same view of the world when an event like a message delivery or membership change occurs (because each process has seen the same preceding set of events and in the same order). Since the programmer

Figure 1.3: An execution in the *ISIS* environment.

knows the algorithm each process follows, he or she can code each process to make unilateral decisions and know that they will all make consistent decisions. No special agreement protocols need be coded (ISIS does all this hard work for you). The result is a program that is simpler to code, easier to understand, and quicker to debug.

One feature of Figure 1.3 is worth elaborating. Observe the delivery of $m3$. At the time P initiated the broadcast, process C was not a member of `Pgroup`. However, by the time delivery occurred, the members had been notified of C's join. The question arises of whether $m3$ should be delivered to C or not. P broadcasts $m3$ to the group `Pgroup`, and if it is treating the group as an abstract entity, it should not be concerned with the actual membership of the group. On the other hand, if a member receives a broadcast message when it has a certain view of the group membership, it is reasonable for it to expect that the message was sent to all the other members in its view of the group. This is precisely what ISIS guarantees. If a member of a process group receives a broadcast message addressed to the group, then a copy of the message will also be sent to all other members that it knows to be in the group at the time the message is received (this membership may be different from when the send occurred). Accordingly, copies of $m3$ are delivered to A, B and C. It follows from this that all the members will have exactly the same view of the group membership when any particular broadcast message is received.

Virtual synchrony

The discussion above should have given you an idea of how the ordering of events ISIS makes it easier to write distributed programs. To enable you to use ISIS most effectively, however, we urge you to make one more conceptual shift in the way you view ordered events. Compare Figure 1.3 with Figure 1.4. In Figure 1.4, each process sees the same set of events and in the same order as in Figure 1.3. In other words, unless a process actually looks at a clock and records the time, *the execution in Figure 1.4 is indistinguishable from the one in Figure 1.3*, at least from the point of view of any individual process. The difference is purely conceptual and lies in the way you, as the programmer, look at the execution. Figure 1.4 shows distributed events like message delivery happening everywhere at the same time (i.e. in synchrony). Of course, ISIS does not guarantee that distributed events will actually be synchronized; it only guarantees that they will be

Figure 1.4: A synchronous execution.

ordered as in Figure 1.3. But as we earlier observed, a process cannot distinguish the ordered execution (Figure 1.3) from the synchronous one (Figure 1.4). What this means is that a programmer can code each process *as if* the execution will actually be synchronous. Any actual execution in the ISIS environment will only be ordered, not synchronized, but a process will not be able to tell the difference (unless of course it records the clock time). So any code written as if the environment were actually synchronous will work correctly when run under ISIS. This is why we call the ISIS execution environment *virtually synchronous*.

Virtual synchrony enables a programmer to write code while thinking of distributed events as occurring everywhere at the same time—precisely the kind of thinking we said was impossible in a disordered environment. To actually make distributed events happen simultaneously would be wasteful and inefficient because this would mean giving up the vast potential for concurrency normally available in a distributed system. Instead, ISIS enforces enough order that the resulting code works correctly, while not sacrificing concurrency. This idea goes beyond message delivery and the notification of events. Each of the ISIS tools is designed with virtual synchrony in mind. Even though they are quite complex and concurrent internally and often involve a number of rounds of communication, they can all be used as if their actions happen instantaneously and indivisibly at all the relevant processes. The state transfer tool described below is one such example. The result of virtual synchrony, then, is to remove from the programmer much of the complexity that arises from distribution, concurrency and fault-tolerance, making it as easy to write a distributed program as it is to write one for a single central machine.

State transfer

Let us go back to our time-service example. The rule we used to divide the work has one big disadvantage. Each time the group membership changed, a member could become responsible for a completely new set of departments. It had to close all its files and open new ones. In a real-life implementation, parts of the file (or all of it) would be read into main memory and fast access structures constructed to search through the data. All this would have to be redone each time the membership changed. Let us instead consider a rule for dividing the work that avoids unnecessary reassignment. For example, we could adopt the rule that when a member leaves the group, its depart-

ments are assigned to the member responsible for the fewest departments. A member that joins the group takes over half the departments from the member with the most departments. To compute the new assignments under this rule, it is not enough for a member to know just its own assignment of departments. Each member has to know the assignments of all the other members as well. The code below shows how this might be done.

```
#include    <stdio.h>
#include    "isis.h"
#define     QUERY_ENTRY     1
#define     MAX_NAMELEN     64
#define     NDEPTS          5
#define     MAX_MEMBERS     10

int         n_assignments = 0;
struct
{
    address his_addr;
    int     his_ndepts;
    int     his_dept[NDEPTS];
}           assignment[MAX_MEMBERS];

address     *gaddr_p;
int         my_index;
int         my_ndepts = 0;
int         my_dept[NDEPTS];
FILE        *my_file_p[NDEPTS];

main()
{
    /* Same as before */
}

service_maintask()
{
    /* Same as before */
}

group_change (gview_p, arg)
```

```
  groupview *gview_p;
  int       arg;
{
    char    filename[16];
    int     n_members;
    int     i, small_i, failed_i;
    address small_addr, *failed_addr_p;

    /* Compute a unique index for this member */
    i = 0;
    while (!addr_ismine (&gview_p->gv_members[i]))
        i++;
    my_index = i + 1;

    /* Record number of members in group */
    n_members = gview_p->gv_nmemb;

    /* Reassign departments from failed members */
    failed_addr_p = &gview_p->gv_departed[0];
    while (!addr_isnull (failed_addr_p))
    {
        /* Find member with fewest departments */
        nsmall = NDEPTS + 1;
        for (i = 0; i < n_assignments; i++)
            if (assignment[i].his_ndepts < nsmall)
            {
                small_i = i;
                nsmall = assignment[i].his_ndepts;
                small_addr = assignment[i].his_addr;
            }

        /* Transfer departments from failed member */
        failed_i = 0;
        while (!addr_isequal (&assignment[failed_i].his_addr,
                                        failed_addr_p))
            failed_i++;
        for (i = 0; i < assignment[failed_i].his_ndepts; i++)
        {
            assignment[small_i].his_dept[
```