ISIS DOCUMENTATION: RELEASE 1*

Kenneth P. Birman
Thomas A. Joseph
Frank Schmuck

87-849
July 1987

TECHNICAL REPORT

Department of Computer Science
Cornell University
Ithaca, New York

12

ISIS DOCUMENTATION: RELEASE 1*

Kenneth P. Birman
Thomas A. Joseph
Frank Schmuck

87-849
July 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

*This work was supported by the Defense Advanced Research Projects agency (DoD) under ARPA order 5378, Contract N00140-87-C-8904, and by the National Science Foundation under grant DCR-8412582. The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE ADA183149	Form Approved OMB No. 0704-0188 Exp. Date: Jun 30, 1986
1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release Distribution Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Kenneth P. Birman, Assist. Prof. CS Dept., Cornell University; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) 4105 Upson Hall Cornell University Ithaca, NY 14853	7b. ADDRESS (City, State, and ZIP Code) Defense Advanced Research, Project Agency Attn: TIO/Admin, 1400 Wilson Blvd. Arlington, VA 22209-2308
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/ISTO; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER ARPA Order 5378 Contract N00140-87-C-8904
8c. ADDRESS (City, State, and ZIP Code) See 7b.	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
ISIS Documentation: Release 1

12. PERSONAL AUTHOR(S)
Kenneth P. Birman, Thomas A. Joseph, Frank Schmuck

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical (Special)	FROM ______ TO ______	July 1987	89

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

ISIS DOCUMENTATION: RELEASE 1 *

Kenneth P. Birman
Thomas A. Joseph
Frank Schmuck

Department of Computer Science
Cornell University
Ithaca, NY 14853

July 1987

* This work was supported by the Defense Advanced Research Projects agency (DoD) under ARPA order 5378, Contract N00140-87-C-8904, and by the National Science Foundation under grant DCR-8412582. The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

Partial contents:

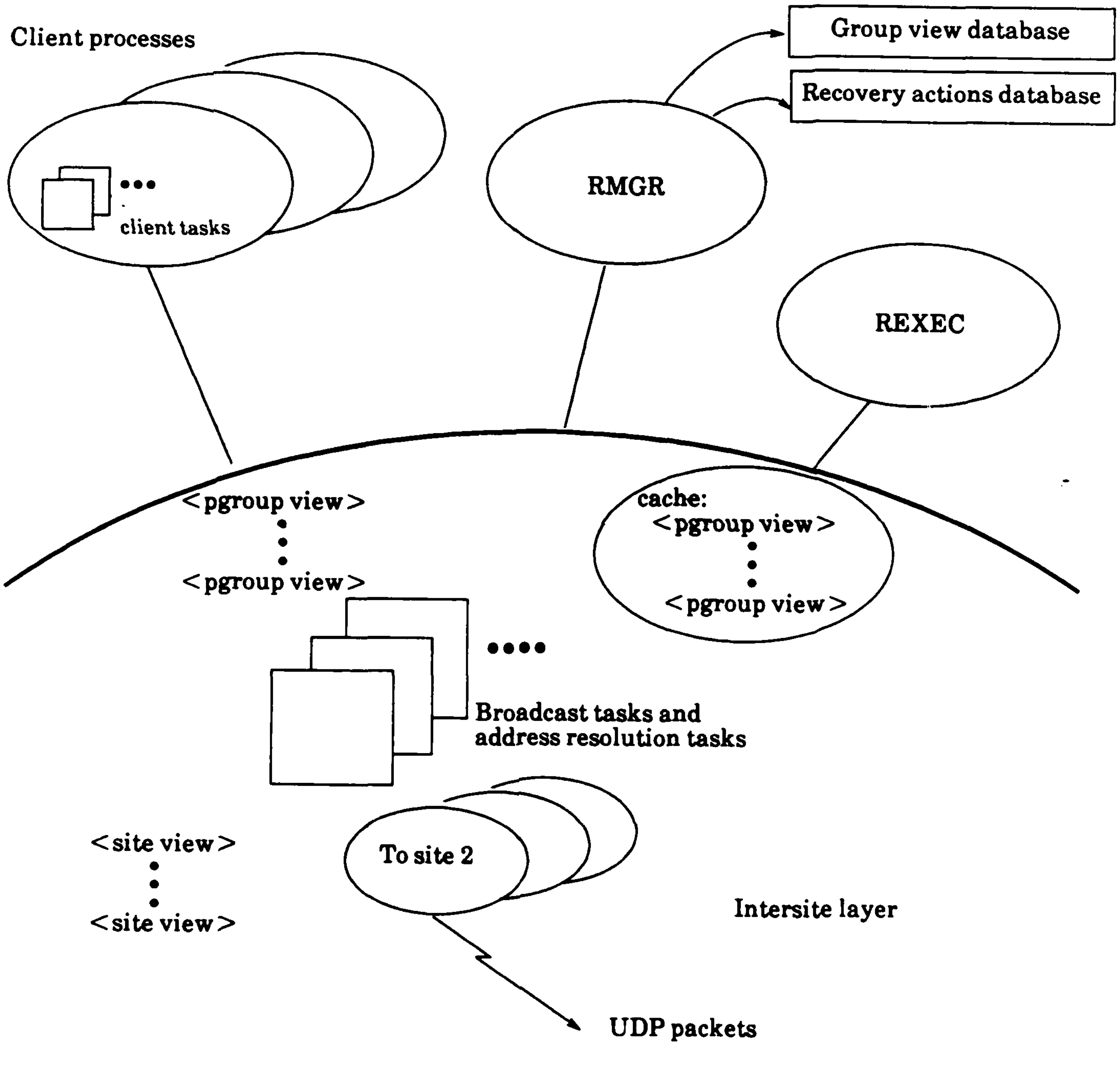
1. List of Documents

The available documentation is as follows:

Addressing -- how processes are addressed in ISIS.
Authen -- a tool used by process to protect against unauthorized requests;
Bboards -- a very high level "bulletin board" facility.
Bcast -- how to do broadcasts in ISIS.
Bitvec -- how bit vectors are used in the site-views data structure.
Config -- configuration data structure manager;
Compile -- how to compile ISIS client programs under UNIX;
Coord -- coordinator-cohort routines.
Entries -- how to define the entry points to a program.
Files -- files the system uses, and how to start the system up.
Filter -- a technical discussion of message filters, not for novices.
Init -- how to call the isis_init() routine.
Logs -- Some suggestions on how to obtain "recoverable" replicated data
Messages -- all you need to know about messages in ISIS.
Msg -- the actual message editing routines, summarized.
News -- a bulletin board facility.
Pgroup -- all about process groups and process group views.
Protection -- overview protection features in ISIS.
Recovery -- a short example on using the recovery manager.
Replication -- a general purpose replicated data management tool.
Rexec -- remote execution facility.
Rmgr -- recovery manager (restarts you after a crash).
Rmupdate -- utility for use with the recovery manager.
Sema -- semaphores for synchronization.
Startup -- a long example on starting up an ISIS program. and
State_xfer -- a state transfer utility, very useful.
Sview -- site views.
Tasks -- a lightweight task mechanism that you currently have to use.
Transactions -- a transaction facility.
Vsync -- a discussion of virtual synchrony and how to exploit it.
Watch -- a facility for watching for a desired (or feared) event.

2. Design philosophy

Although these documents are presented in alphabetical order, there is a good order in which to read them in. Before starting, we recommend that you learn a little about ISIS. Some of the recent papers would be fine, or one of the short overviews we have generated over the past few years. Once you think you have the picture, start by reading about addressing, messages, tasks, process groups, and entries. Then read about compiling, system files, the init routine, and the thing called "startup", which describes a long example. Documentation on the various toolkit routines can be read as needed. The most useful ones are probably rexec, rmgr, state_xfer, coord-cohort, sema, and update. Most toolkit mechanisms are orthogonal to the others, although this may be hard to see until you get some experience using the system. Also, some work will be needed on parts of the system concerned with recovery before this can be done transparently. Thus, a good development strategy is to start with the system design for the "normal" case, then add code for recovery from failures and dynamic reconfiguration -- it won't change your original code much at all. The evolution of the twenty questions program is illustrative of this design philosophy.



ISIS at a site

1. Synopsis

A discussion of addressing in the ISIS system.

2. Include files

```
#include <isis/cl.h>
```

3. Type definitions

From the bottom up, ISIS knows about *sites*, *site lists* and *views*, *process addresses*, *group addresses*, and group *views*. There is also a notion of *remote* sites, processes, and groups, which can only be accessed using special protocols.

1. Currently, ISIS only supports *local sites*. A *local site* is identified by a two-byte sequence consisting of a site-number, in the range 1-127, and a site-incarnation, also in the range 1-127. This sequence is referred to as a site-id in the ISIS system. In future versions of the system, site-numbers will be expanded to include a concept of local sites, long-distance sites, and remote sites. In this extension, site-id's will be 4 bytes long. Two bytes will represent the *cluster* number, one byte the site-number, and one byte the site incarnation number. Local sites will have a cluster number of 0. Long-distance sites are intended to represent sites within the same geographical area but accessible at somewhat higher cost, e.g. through a gateway. The cluster number for these sites will be in the range 0-127. Remote sites are assumed to be accessible only over genuinely long-distance connections and will use a hierarchical numbering scheme. A remote site number will be represented by cluster number in the range 128-255 and must be mapped through a *mount* table to obtain remote addressing information, using a method that is at present unspecified. In most cases special protocols will be used to communicate with remote sites. Process groups will not be allowed cross long-distance communication boundaries, but mechanisms for linking copies of a group that lives on both sides of such a boundary will be provided.

   The number of sites in a cluster is intentionally kept small to control the costs of the ISIS protocols. The actual decomposition of sites into clusters is transparent to the ISIS user, but can affect performance: whenever possible, processes that interact heavily with one-another should be located within the same cluster. In addition, it is undesirable for clusters to "partition" in such a manner that communication between two subgroups of the cluster is temporarily impossible. For example, if a cluster contains a single gateway, ISIS may block (hang) during periods when the gateway is down. This problem can be circumvented by introducing redundant communication gateways whenever possible.

   Several pre-defined macros allow one to extract the fields from a site_id sid: SITE_NO(sid), SITE_INCARN(sid) and SITE_CLUSTER(sid). The macro MAKE_SITE_ID(site-no, incarn) can be used to create a local site-id.

2. A site-list is a list of site-id's terminated by a null site-id. Note that SITE_NO(sid) is null only for a null site-id. This is useful when scanning the elements of a site-list.

3. A site view consists of a list of site-id's and associated information maintained by the system failure detection module for a single ISIS cluster. In particular, a view has a view-id number, and all sites in a cluster observe the same sequence of views. It can be assumed that the sites in a site-view are listed in order of age (oldest first) and that all observers see the same sequence of site-views. See SVIEW(TK) and VSYNC(TK) for details.

4. A process address in ISIS consists of a site-id, a type field containing the constant ISAPID, a unique process-id number which is a short integer used by the operating system at that site to identify a process running on its site, and an entry point within that process, which may be null. The procedure MAKE_ADDRESS(site,incarn,pid,entry) can be used to make a process address. The corresponding field names are *site*, *incarn*, *process*, *entry*. The site

field will never be null, hence this is sometimes used to detect the end of a list of addresses. Certain predefined id-numbers are used to identify system processes. For example, the defined symbols PROTOCOLS, REXEC, RMGR, and NEWS are automatically mapped to the process-id for the corresponding service at a given destination site. Additional system-wide process numbers will be added as the ISIS system evolves. These addresses are defined in generic_address.h.

The function cmp_address() is provided to facilitate address comparisons. Invoked as cmp_address(a1,a2), where a1 and a2 are pointers to addresses, this returns 0 if a1 and a2 are the same, a negative number if a1 and a2 differ and a1 is "smaller", and a positive number if a1 is larger. Thus, although most users would just compare the result with 0, cmp_address() is compatible with the standard UNIX quicksort() utility. The caller of cmp_address should be aware that an address with the entry field specified as 0 is treated specially: such an entry is a wild-card that will match any other entry value. Two addresses with non-zero entry numbers must match exactly, however.

5. A group address is an address used to identify a process group in ISIS. Such an address consists of a site-id for the site, a type field containing the constant ISAGID, a 16-bit group id, and an entry point *that must be the same for all members of the group*. Notice that group and process addresses both have the same format; if desired, a process address may be thought of as a group containing one member. Group addresses are created using the pg_create() request, but because of subsequent join, leave and failure events the group may subsequently migrate to other sites in the system. Consequently, group addresses are usually obtained using pg_lookup(). This implies that the site-id in the group address is not necessarily useful for determining where members of the group reside (but see also PGROUPS(TK)).

4. Entry points

Each process in the ISIS system is understood to accept messages at a variety of entry points. An entry point is a one-byte unsigned integer.. Some entry points have standard values: GENERIC_RCV_REPLY is the entry point to which a reply message can be sent, GENERIC_TK_CHKPT is the entry point used by the checkpoint toolkit routine to trigger a checkpoint, etc. These are defined in generic_address.h, which is automatically included when "cl.h" is included into a program. In addition, each process can define additional entry points of its own. To avoid accidental conflict with these generic addresses, these user-defined entry points should be assigned entry numbers greater than or equal to USER_BASE, a constant also defined in that file. Notice that different processes can interpret the same entry "number" in different ways.

A process declares its entry points by calling the routine

```
isis_entry(entry-point-number, routine, "printable name");
```

Many of the toolkit routines install their own handlers (for the GNEERIC entries) when isis_init() is called. On arrival of a message, the corresponding entry will be invoked as:

```
routine(mp)
  message *mp;
  {
  }
```

The message is automatically deleted after the routine terminates, unless msg_increfcount() has been called prior to returning. If a message arrives in a process and the process has not specified a routine to handle messages to the specified entry point, the message is discarded and an error message is printed on the stderr output channel.

See PGROUP(TK) for information on manipulating process groups, ENTRY(TK) for more information on entry points, MESSAGES(TK) for more information on messages, and BCAST(TK) for information on sending messages to the members of one or more process groups.

5. RPC interactions

An RPC style interaction occurs when a process sends a message to another process then awaits a reply. ISIS supports this mode of interaction, and will even provide stub generators to compile from a "nice" looking RPC syntax into the message generation and unpacking mechanisms needed to map this into the above facility. To identify the RPC "session", a session-id number is placed in the message at the time it is sent (see MESSAGES(TK)). The sending task then blocks awaiting a reply with this session-id number; session-id numbers are 32-bit integers and should not be re-used. Thus, a pending RPC has an address consisting of the address of the caller process together with the id of the session. To send a reply, the replying task creates a message containing the reply value (field name FLD_ANSW), the length of this field (FLD_ALEN) and the session-id number identifying the session (FLD_SESSION), and then transmits this message to the sender of the RPC. In general, ISIS does not assume that it is an error to send the same reply more than once or to send multiple replies to a task that expected just one reply. In these cases, the superfluous replies are discarded silently.

6. Printing an address

The routine paddr(addr) will print the address pointed to be addr; paddrs(alist) will print the members of a null-terminated address list, and psite(sid) will print the site name and incarnation for a site=id. Whenever possible, entry numbers are printed in their text form, but if paddr() is called in a place that just doesn't know the text form for an entry point, the numeric version is printed instead. This is true for process-id's too.

7. Site names

The array site_names[] gives, for a site-id, the printable name of that site. These names are actually taken from a file used during startup of the system (see FILES(TK), STARTUP(TK)).

1. Synopsis

A mechanism for restricting access to a group.

2. Interface

```
#include <isis/cl.h>

        au_request_verify(routine)
          int (*routine)();

        au_permit(who)
          address who;

        au_revoke_perm(who)
          address who;
```

3. Discussion

Normally, any process in possession of the address of a group can issue calls to that group. Some applications will need more protection than this, however, and the authentication tool gives them that option.

To enable the tool, call au_request_verify(routine), giving a routine that will verify the legality of requests from unknown callers. The routine is invoked as:

```
        routine(mp)
          message *mp;
          {
          }
```

and should return 0 if the request is legal. A reply(mp,0,0,0) is sent by the authentication service if the routine returns -1. No reply is sent if the routine returns some other value. To avoid unnecessary work, the routine au_permit() can be called to indicate that the designated caller is permitted to send arbitrary requests to this process. All messages from that process will be allowed through. au_revoke_perm() removes an address from the privilaged caller list; subsequent messages from that process will be passed through the verification procedure.

4. Restriction

The verification procedure is permitted to call t_fork_delayed, t_fork_urgent, t_sig_delayed and t_sig_urgent, but may not call t_wait or try to do an RPC or broadcast. This is because it is run from the main thread of control -- not as a task.

1. Synopsis

A package of routines implementing distributed bulletin boards as described in our technical report. These routines will be implemented during the summer or fall of 1987. The interface will be a subroutine one but otherwise very similar to the one discussed in the paper. Initially, only C will have access to the bboard facility, but versions for other languages (especially LISP) will be provided eventually.

1. Synopsis

A package of routines implementing distributed broadcasts of various flavors, and with a variety of destination addressing modes.

2. Interface

```
#include <isis/cl.h>

        isis_init(0);

        . . .

        /* Broadcast to a list of addresses */
        nresp = BCAST(alist, msg, nwanted, answ, atype, alen, rlist)
          address *alist, *rlist;
          message *msg;
          char *answ;

        /* Broadcast to everyone on a list except the sender */
        nresp = BCAST_EX(alist, msg, nwanted, answ, atype, alen, rlist)
          address *alist, *rlist;
          message *msg;
          char *answ;

        /* Reply to a broadcast or RPC request */
        reply(msg, value, type, len)
          message *msg;
          char *value;

        /* Reply, sending a copy to other processes */
        reply_cc(msg, alist, value, type, len)
          message *msg;
          address *alist;
          char *value, *fvalue;

        /* Flush any asynchronous messages */
        FLUSH()
```

Above, BCAST is an unordered (but reliable) protocol, and is not actually used very often in ISIS. You can substitute CBCAST to obtain the causal broadcast, ABCAST for the atomic broadcast, and GBCAST for the strongly ordered group broadcast protocol. Section 4, below, discusses the way this choice would normally be made.

3. Discussion

In each case, the addressing information is used to determine a set of destination processes (an address list) to which the message is delivered. On reception of a message, this information will be present in its *dests* field (see msg_getdests() in MSG_EDIT(TK)). The protocol waits until *nwanted* replies are collected, or until it has as many replies as possible, and then returns the number of replies and a vector containing the replies themselves. The address of the sender who supplied the i'th answer will be saved in rlist[i] if rlist is non-null, and is discarded otherwise.

If *nwanted* is 0, the message will be sent asynchronously. That is, the caller can continue executing before the message is delivered, although there will be a delay even in this case while the message is passed to the ISIS protocols process. A message is said to be synchronous if the caller blocks waiting for one or more replies, although obviously there is a range of degrees of synchrony: a caller that waits for one reply will be running much more asynchronously than one that waits for replies from all destinations. Asynchronous execution is always much faster than synchronous execution; the more synchronous, the higher the performance cost.

A reply is specified as a pointer to the data in question (it will be copied to a safe place during the call), the type (see MESSAGES(TK)), and the length of the data item in bytes. Replies to a message are sent using reply() or reply_cc() (which sends copies of the reply to some other set of processes). If the caller has specified ALL for *nwanted* it is still possible for a recipient of the message to refuse to reply; this is done by calling a reply with a null *answ* pointer and a 0 *alen*. The reply is also permitted to be shorter than the length specified in the broadcast.

For example, the fancy twenty-questions program described in the XFER(TK) documentation uses a reply(mp,(char*)0,0,0) when one of its hot standby processes gets a request message.

The addressing rules used by the broadcasts are relatively subtle. The basic idea is this:

a) BCAST() sends to the processes and process group members listed in the null-terminated address list. It is assumed that the "entry" field of each address in the list has been set to the entry number to which the message should be delivered (see ADDRESSING(TK)). If this is a standard entry and hence the entry number will always be the same, the routine set_entry(alist,value) can be used to set all entry numbers in the alist to the designated value (for convenience, set_entry returns its alist argument).

b) BCAST_EX() is like BCAST(), except that if the sender is a member of the address list it will be excluded from the actual delivery. This is useful when an asynchronous broadcast is to be sent to the remote managers for some distributed resource after the local copy has already been updated.

What makes addressing complicated is that ISIS makes a distinction between process groups that are directly accessible by a process and those that it can only access indirectly. A group is *directly accessible* by any of its members, plus any additional processes that a group member has added to the group view using pg_addclient() (see also PGROUPS(TK)). Alists as described above can only be used if all the process groups in the alist are directly accessible (other processes may be explicitly listed too). *If a message is to be sent to a process group that is not directly accessible, the alist must only contain one entry -- the group address*. Thus, broadcast addressing is far more flexible in the case of directly accessible addresses. To make matters worse, CBCAST() doesn't work correctly if invoked asynchronously from a process that can only access the destination indirectly. Thus, in the case of indirect access, CBCAST should only be used synchronously (waiting for responses from one or more destinations). This limitation will be eliminated in a future release of ISIS.

If a broadcast is invoked with *nwanted* equal to 0, or if several broadcasts are done concurrently by different tasks within a single process, the issue arises of how to ensure that they have terminated before taking some action that might leave an externally visible trace. Otherwise, should a failure cause one of these protocols to abort, the external state of the system might be inconsistent with the state left by the failure. The FLUSH() primitive should be invoked for this purpose. It blocks until all pending broadcasts are completed and then permits the caller to resume computation normally.

4. Picking the right flavor of broadcast.

In most cases, CBCAST should be specified as the broadcast primitive; this is the cheapest protocol in ISIS and it is highly advantageous to use it whenever possible. However, some replicated data structures and algorithms need the stronger ordering that ABCAST and GBCAST provide, and there is no very simple way to explain how one identifies these applications. The basic rule is: CBCAST is used when messages from other processes that happen to arrive at the same time as your broadcast will be serviced the same way regardless of whether your message arrives first or second. For example, requests to read a replicated database or for some other simple service would be transmitted using CBCAST: these have no effect at all, and databases are usually locked to prevent reads while they are being updated. ABCAST is used when requests will be queued or otherwise applied to a replicated data structure that would return different results after a sequence of updates depending on the order in which they were done. A FIFO queue has this behavior, but a B-tree or a file normally would not. Thus, one would normally use ABCAST when talking to a FIFO queue and CBCAST when talking to a B-tree manager. GBCAST is used to obtain a consistent cut across the system, an operation that is only needed in certain highly sophisticated algorithms. If you are concerned that your application may have an order sensitivity, it should still suffice for you to use ABCAST. ABCAST, however, is slower than CBCAST. More discussion of this choice appears in VSYNC(TK).

5. RPC mechanism

ISIS does not currently support the sort of argument packaging that is common in RPC services such as the SUN RPC service or the CEDAR-MESA one. However, it will shortly. In the meantime, to do an RPC, the arguments should be packaged in a message, and then one of the broadcasts used to transmit this message to its destination (that is, the alist should specify a single destination). Set *nwanted* to 1 and wait for the answer. It is easy to generalize this to a group-RPC (set *nwanted* to ALL) or a quorum-oriented RPC (set *nwanted* to the quorum size). The RPC will return when nwanted responses are received and any extra responses will be discarded. Within ISIS itself, we use a combination of these methods.

6. What can you conclude about a failure?

If a broadcast routine is asked to wait for a reply from some process, but it returns without that reply, the process has failed or simply doesn't exist. You can actually conclude a bit more about the system state than this, however, and understanding what you can assume will simplify your code.

First, it is safe to assume that you won't see any more actions or messages initiated by the failed process. For example, if it was supposed to send a reply (or a reply_cc), either the reply gets through first, to all destinations in the reply_cc case, or the reply just won't arrive anywhere, ever. This is also true when the process might have been using a toolkit routine at the time it crashed. For example, if a process may have failed while pg_addmemb() was running, either the new member will have been added before the failure is detected, or the failure will be detected first in which case the pg_addmemb() will not take place -- this is because the addmemb() algorithm in ISIS is based on GBCAST, and either the GBCAST is delivered before the failure is announced, or not at all.

Thus, if you didn't get some message and the process that is supposed to send it is observed to fail, you won't get it -- and will did any one else who would have received a copy. This is a property called *broadcast atomicity*.

Also, the failed process will vanish from any process group views in which it was listed, and if the broadcast used one of these groups as a destination, the failed process drops out before the broadcast returns to its caller. The process may not, however, have been dropped from other groups to which the message was *not* sent. This is becsuse there could be a delay between when view one gets changed and when another does.

For example, assume that process 'p' is a member of groups g1 and g2. Some other process, 'q' broadcasts to g1 and waits for replies from all its destinations. If 'p' fails, process 'q' will definitely find that 'p' has dropped from group g1. On the other hand, 'p' may still be listed as a member of group g2, which was not a destination of the broadcast.

1. Synopsis

Some simple routines for manipulating vectors of bits.

2. Interface

```
#include "cl.h"

# define      MAXBITS        32      /* Multiple of 32 */

        bis(vec, b)
          bitvec vec;
          int b;

        bic(vec, b)
          bitvec vec;
          int b;

        bit(vec, b)
          bitvec vec;
          int b;

        bisv(vec1, vec2)
          bitvec vec1, vec2;
          int b;

        bicv(vec1, vec2)
          bitvec vec1, vec2;

        bandv(vec1, vec2)
          bitvec vec1, vec2;

        bitv(vec1, vec2)
          bitvec vec1, vec2;

        bclr(vec)
          bitvec vec;

        bset(vec)
          bitvec vec;
```

3. Discussion

These routines support vectors of bits of arbitrary length and are used by ISIS to implement the sv_failed and sv_recovered parts of a site-view. They implement 32-bit vectors as long integers and longer vectors as character arrays. They can be used for other purposes, but you may be forced to use the same value for MAXBITS as the remainder of the system if you include cl.h in the file that employs the bitvectors. The routines will set/clr/test a single bit, set/clear/and/test bit by bit between two vectors, clear an entire vector (set bits to 0), or set an entire vector (set its bits to 1).

1. Synopsis

How to compile an isis client program.

2. Synopsis

```
ln -s ~isis/client/lib1.a l1
ln -s ~isis/client/lib2.a l2
ln -s ~isis/msg_edit/mlib.a l3
cc -c -Iisis/client client_prog.c
cc -o client_prog client_prog.o l?
```

3. Warnings

ISIS uses a number of global variables, and it is obviously a bad idea to re-use the same variable names for some other purpose. We try to use names like cl_... or pr_.... to avoid likely conflicts, and to declare our variables to be static whenever possible, but some care is certainly required. Many of these global variables are declared in cl.h. Eventually, we plan to clean this up and will also provide a list of global variables and what they are used for below.

Shortly, the library called mlib.a will be merged with lib2.a. The use of two libraries is an unavoidable consequence of the way RANLIB works. The first (lib1.a) contains toolkit routines, and the second contains the remainder of the client->isis interface code.

3.1. Temporary SUN version

We have some ideas on how to reorganize the system, but for now the various libraries come in two versions. The one shown above is for the gould. On the SUN, everything is the same except for the client directory, which is renamed "klient", and the message edit library, which is renamed sun_mlib.a. This situation will go away very soon.

1. Synopsis

A toolkit routine for managing configuration information

2. Interface

```
#include <isis/cl.h>

        config_update(gid, name1, data1, type1, len1, name2, ... , 0);
          address gid;
          char *name1, *data1;
          int type1, len1;

        char *config_get(gid, name)
          address gid;
          char *name;
```

3. Discussion

Some applications will need to divide up tasks using application specific rules that change dynamically. The configuration tool makes this easy, requiring only that the configuration updates be done by members of the group to which the configuration applies, not outside "clients". The rule should be represented using one or more data structures; multiple structures would be used in some cases because of the need to specify "type" information using the mechanism supplied by the message editing routines (see MESSAGES(TK)). To update the configuration structure, use config_update. When a message arrives, all recipients should call config_get to query the structure and all will see the same values in it for any given message. You should copy these values to the side if you plan to look at them after doing something that could block (an RPC or broadcast, a t_wait(), etc); these values can change while a task is asleep. Configurations are maintained on a per-group basis; the same field may have different values and meanings for two different groups even if the some program belongs to both groups.

It is costly to update configurations, especially if the same configuration is updated concurrently by multiple group members. Therefore, such behavior must be avoided whenever possible.

1. Synopsis

A routine implementing coordinator-cohort computations.

2. Interface

```
#include <isis/cl.h>

        isis_init(0);

        . . .

        /* Run a coordinator-cohort computation */
        coord_cohort(msg, gid, alist, action, rtype, rlen, got_reply)
          message *msg;
          address gid, *alist;
          char (*action)();
          int (*got_reply)();

        /* Figure out who the coordinator should be */
        address coordinator(gid, sender, alist)
          address gid, sender, *alist;
```

3. Discussion

Many ISIS algorithms are based on the idea of having one process (the coordinator) take some action while others (its cohorts) monitor it and take over in the event of failure. Although this can be implemented several ways, we picked a simple scheme and provided it in the coordinator-cohort toolkit facility. Since the notion of picking a coordinator for a task is somewhat more general than the notion of running a coordinator-cohort computation, the routine we use to pick the coordinator is also documented here.

A coordinator-cohort interaction starts when a client issues a request to some group of processes using a broadcast. The client typically blocks waiting for a single reply, which may come from any of the destination processes. The recipients of this message *all* invoke coord_cohort with the following arguments: the message, the address of a routine that will take the desired action, the processes at which the computation is running, and the length and type of the result (see MESSAGES(TK) and MSG_EDIT(TK)). The list of processes should be in the same order at every process that invokes the coord-cohort routine; this is facilitated by the fact that the message destinations (see msg_getdests()), the members of a process group (pg_getview(gid)->pg_alist) and the sites in a site-list (sl_getview()->sl_slist) have the same values in all of these processes when the message arrives. See VSYNC(TK) if this concept confuses you. Basically, the idea is that messages in ISIS seem to arrive simultaneously at all destination processes.

The coordinator site will be the site where the message was sent if one of the processes in the alist resides at that site, and randomly chosen otherwise. The other processes are cohorts and are ranked using a fairly random algorithm based on site-id numbers. The processes in the alist must all be members of the group designated by gid.

At the coordinator, the action routine is invoked as action(msg,gid,how), where gid is the group id from the coord-cohort request and *how* will be the constant CC_COORD, defined in <isis/cl.h>. The coordinator routine should execute the request and compute a result, storing it in an area of memory allocated with malloc. It should then return a (char *) pointer to this area. This result will be sent to the caller using a reply() mechanism but will also be transmitted to the cohorts, where the got_reply routine is invoked as: got_reply(msg, result, rlen). The type field is used in generating the reply message, but is not passed as an argument to the got_reply routine. The msg argument is the one to the original coord_cohort call. The memory that the result occupied is automatically freed when no longer needed.

In the event of a coordinator failure, one of the cohorts will take over and restart the action. The restart invocation is identical to the initial action invocation except that *how* will now be equal to CC_COHORT. No clean-up actions will have been taken; the cohort is responsible for this if any are needed. If all of the processes in the alist fail, the caller receives a failure indication from the original BCAST() that triggered the execution of the algorithm -- specifically, the BCAST() returns 0 (no replies) instead of 1 (the single reply the caller wanted).

The routine coordinator(gid, sender, alist) picks a coordinator and returns its address. It returns NULLADDRESS if every process in the alist is down. The coordinator will be an operational member of *alist* in the current view of *gid* subject to the following rule:

1. If some process in the alist is at the same site as the sender, the coordinator will be picked relative to this process.
2. Otherwise, the coordinator is picked relative to process alist[k] where k = sender.site mod length(alist).
3. Now, given a starting point, entries in alist are evaluated one by one, and the first one that is listed in the current view for *gid* is returned. NULLADDRESS is returned if all processes in alist are tested and none is operational.

1. Synopsis

Declaring the routine that will service requests to a given entry point.

2. Interface

```
#include "cl.h"

        isis_entry(code, routine, rname)
          int code;
          int (*routine)();
          char *rname;
```

3. Discussion

When starting up, a program should bind routines to the entry codes that it will accept in messages it receives. The toolkit routines do this for the generic addresses when isis_init() is invoked. Once defined, it is illegal to redefine the *generic* entry points, although user entry points can be rebound as desired. This is to prevent users from unintentionally screwing up the toolkit routines. Entry points are bound by calling isis_entry and specifying the numeric code, the routine address, and a printable name corresponding to this routine. The generic entry codes are defined in generic_address.h; these are standard for all ISIS clients. Other entry codes can be defined on a per-client basis starting with the number USER_BASE. Codes need not be allocated sequentially and different applications can use the same entry code for totally different purposes.

4. Intercepting a message

It is possible to intercept and examine messages before they reach the entry handling routine. See FILTER(TK) for details.

1. Synopsis

Description of files used when starting ISIS up at a site and the program used to start the system up.

2. File summary

sites: Lists the sites that are running this time
restart: Tells what programs to restart automatically

3. Starting isis up at a site

To run isis at a set of sites, first create a "sites" file in the following format: a '+' or a '-' (lines with a minus are ignored), site-number (must start with 1), a colon, three numbers indicating the internet ports for the isis sites to talk to each other, to use when restarting, and to talk to clients, the site name, and if multiple isis systems are run on one site, a '/' followed by a number. The port numbers can all be 0 if the /etc/services file is set up to support isis for your system. The third of these numbers is the one isis_init() expects to be passed. For example, a typical sites file might contain:

```
+ 1:1250,1251,1252 bullwinkle.cs.cornell.edu
+ 2:1250,1251,1252 kama.cs.cornell.edu
```

This file says that isis will be run with two sites operational, bullwinkle and kama. The port numbers used are the same in this case, because bullwinkle and kama are two different machines. To run two instances of isis on bullwinkle use a sites file like this one:

```
+ 1:1250,1251,1252 bullwinkle.cs.cornell.edu/1
+ 2:1253,1254,1255 bullwinkle.cs.cornell.edu/2
```

Here, the port numbers had to be different because all the ports are to be used at one site.

The restart file tells what programs to restart when isis comes up at a site. These are mostly system services, like the remote exec service. Here is a typical restart file:

```
/fs/moose/b/isis/protos/protos <isis-protos> -p
/fs/moose/b/isis/client/rexec <isis-rexec>
/fs/moose/b/isis/client/rmgr <isis-rmgr>
```

This tells the system to run the protocols process (protos), the rexec program (rexec), and the recovery manager. The argument "-p" to the protos process is required when isis is run this way. Extra arguments may be supplied by the isis startup program: in the case of the protos program, -Sname if the sites file name is not the standard one, -H/# if the site has a sub-number, and in the case of the other programs, a port number if the sites file specifies something for the client-isis connection port. For example, using the second sites file and the above restartfile, the protocols program for bullwinkle/2 will be invoked as

```
<isis-protos> -H/2
```

and the rmgr for bullwinkle/2 will be invoked as

```
<isis-rmgr> 1255
```

The latter command is telling the rmgr program to call isis_init(1255) to connect to isis-protos.

To run isis at a site, type

```
isis [-Sname] [-Rname] + &
```

If the sites file name is not given, "sites" is assumed. If the restartfile name is not given, "restartfile" is assumed. The "+" argument tells isis to bypass the site-failure detector, which is not yet fully debugged; in this case, all sites must be brought up more or less at the same time (so, if using bullwinkle and kama, you would have to run isis on both machines at the same time, by hand!). This command will bring up as many instances as are needed for the local host, so using the second site file it will automatically start isis up twice. A SUN/2 begins to perform poorly when it must support more than 2 isis sites at a time, the gould somewhat more (4 to 6 maximum).

A typical program, say 20-questions, would be run as

```
twenty client-port
```

giving the client port-number to use to connect to isis, or just

```
twenty
```

if the /etc/services file is set up correctly to know about isis. *Note: It takes about 30 seconds to restart isis at a site.* If a program tries to connect to isis before restart is finished, various errors can result.

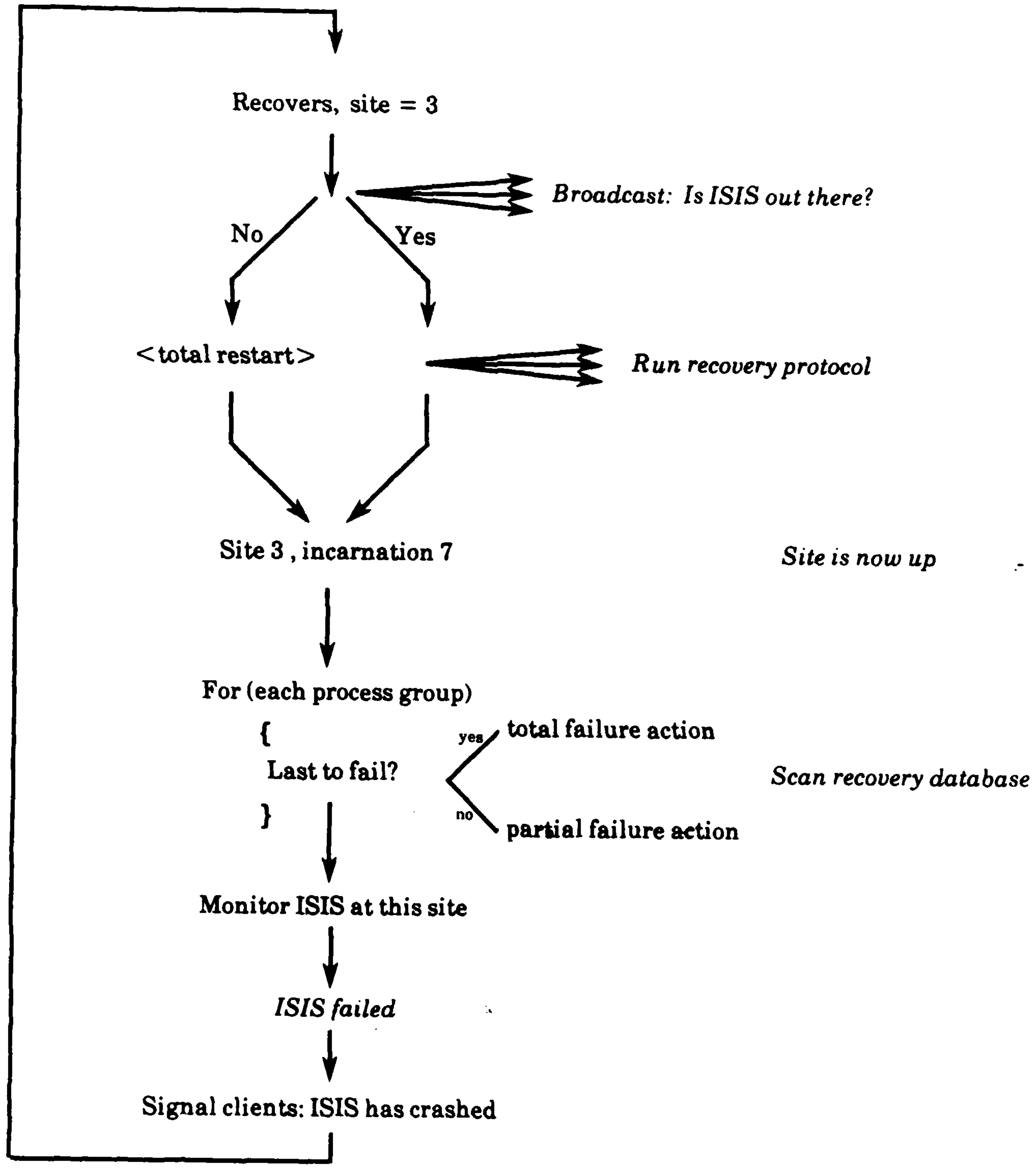
To kill isis at a site, do a

```
ps x
```

and kill the protos process(es) that are listed. If the system crashes, the reason it crashed is usually given in the file isis.log (1.log, 2.log, etc if several run on one system). After killing isis, a UNIX bug can cause the ports it was using to linger for 30-seconds to a minute. If you wait long enough before restarting the system, these will normally go away. If they don't there is nothing to be done except to change the ports isis is using or to reboot the machine. (This is obviously a UNIX bug.)

4. Restart sequence

The figure below outlines the stages of restarting isis at a site. As seen in the figure, "isis" starts by scanning the sites file and then trying to contact isis processes elsewhere in the network. If it finds any, the system restarts by joining them. Otherwise, it assumes the restart is from a total failure. Next, all the standard system programs are run, and then the recovery manager restarts user programs (see RMGR(TK)).



The ISIS Recovery and Monitoring Sequence

1. Synopsis

A mechanism for intercepting arriving messages, used internally by the system.

2. Interface

```
#include <isis/cl.h>

typedef int     ifunc();
ifunc           *isis_setfilter(), old_filter;

        old_filter = isis_setfilter(routine);

        . . .

        (void)isis_setfilter(oldfilter);
```

3. Discussion

ISIS has several tools that "filter" the stream of messages arriving at a process. For example, the state transfer tool spools messages of several types during transfers, the authentication tool validates the legality of arriving requests, etc. A filter works by interposing itself between the arriving message queue and the next "lowest" level filter, down to cl_local_delivery, which is the lowest level of all. A filter is called as a message arrives, and can inspect it:

```
routine(mp)
  message *mp;
  {
  }
```

Actions available to a filter are to reject the message (usually by sending some sort of reply, perhaps a reply(mp,0,0,0), or to pass it to the next level by calling old_filter(mp). A filter may fork off a new task or issue a signal, but must not try to wait or do an RPC or broadcast. This is because a filter does not run as a normal task.

4. Example

The state-transfer utility spools some messages by setting itself up as a filter. Its filter routine looks like this:

```
xfer_filter(mp)
  register message *mp;
  {
      if(<let mp through>)
          old_filter(mp);
      else
      {
          if(xfer_queue == (queue*)0)
              xfer_queue = qu_null();
          qu_add_mp(xfer_queue, 0, mp, nullroutine);
      }
  }
```

When a transfer finishes, messages are despooled by restoring the old filter and then replaying the messages into it:

```
xfer_despool()
```

```
{
    register queue *qh, *qp;
    ifunc filter;
    (void)isis_setfilter(old_filter)
    filter = old_filter;
    qh = xfer_queue;
    xfer_queue = (queue*)0;
    while(qp = qu_head(qh))
    {
        filter(qp->qu_mp);
        qu_free(qp);
    }
    qu_free(qh);
}
```

Notice how the replay mechanism "steps to the side" by using a copy of the queue pointer and reseting the old value to null, and also by reseting the isis filter *before* despooling any messages. This is necessary to ensure that a reinvocation of the state transfer tool (or some other entry point that changes the filter) can set up the filter again and create a new transfer spool. If this idea (a reentrant procedure) is unfamiliar to you, it is probably not a good idea to try and use message filters in your application: the sorts of bugs that you may run into are pretty bizarre!

1. Synopsis

Isis initialization routine.

2. Interface

```
    #include "cl.h"

            isis_init(port-number);
```

3. Discussion

This routine connects the caller program to ISIS and initializes both the data structures used by the task and toolkit facilities and various constants, such as my_process_id, my_address, my_site_no, my_site_incarn, and the site_names[] table (site_names[i] is a printable name for site i). The port-number may be given as 0, in which case the value in the system-file /etc/services is used, or as a non-zero number in which case the specified port is assumed to be the one to connect to ISIS on. See ADDRESSING(TK) for a discussion of the sites file which contains the port number specification. STARTUP(TK) gives some examples of programs that call isis_init, and FILES(TK) talks about where these port numbers come from.

System services like REXEC, RMGR, etc. should set the variable my_process_id to their "name" before calling isis_init.

The task routines may crash if called before isis_init() is called.

1. Synopsis

How to manage replicated data structures that can be recovered after total failures (all process group members die).

2. Discussion

One problem confronted by the ISIS programmer relates to the recovery of a replicated data structure after a failure. In a situation where someone survived the failure, this is easy: the state transfer tool can be used, with the recovery manager giving advice on when to do the transfer. But, what if everyone crashes? The recovery manager will wait until all the relevant sites come back up, then restart the failed processes, but exactly how should the missing data structures be rebuilt?

In many systems, the solution to has been to resort to transactional files on disk. You can do this in ISIS too (see TRANSACTIONS(TK)). Transactionally updated files survive failures and are updated atomically, hence anything damaged by a crash will automatically be restored. In ISIS, however, the entire idea is to move away from nested transactions and towards less costly, less intrusive mechanisms.

A simple way to deal with recovery, and the one we recommend, is as follows: Replicated data is maintained in core, in a volatile form, using tools like the replicated update tool to query and update it. When an update changes the structure in a significant way, however -- a way that needs to be preserved after failure -- the update routine should "log" the change in a well-known file in a well-known place. For example, you could create a file called "/usr/spool/logs/my_prog.log". Use a simple file format (ascii is nice) and keep appending to it and doing a flush (fsync(2)) after each write. On recovery, you will need to reread the file and rebuild the data structure one change at a time.

One thing to be aware of is that when several processes die at once, they could leave their logs in slightly different states. Since the recovery manager arranges for all the last processes to fail to recover at the same time, your recovery mechanism should avoid a situation where these processes recover independently from logs that could have different lengths or contain non-indentical information. Some ways to do this are: have a coordinator recover from one of the logs and then use a state transfer to copy this information to other processes, or maintain the logs in a cannonical order and reach agreement on what the last record is, or maintain identical logs and reach agreement on what the length is. To reach agreement, use a group RPC that queries the other processes with copies of the log and take the minimum length that is returned as the answer.

Frankly, unless your application uses ABCAST whenever an update to the in-core storage occurs, which makes it trivial to keep the logs in a cannonical order, we recommend that you go with the coordinator-cohort solution. It just isn't worth going to so much trouble to deal with a situation that almost never happens anyway. Even this solution won't be trivial, because you have to cover two cases: one where the coordinator succeeds in loading its log and other processes just do state transfers to join, and one where the coordinator dies while loading the log or during a state transfer, forcing a cohort to restart from its own version of the log. A tool that does this will be provided as part of the recovery manager later this summer, and an example that uses it will be included in the next edition of this documentation. The code isn't nearly as complex as it probably sounds.

If your logs are likely to get very long, a periodic checkpoint might be a good idea. You can create one by supporting a "checkpoint" log record, which would always appear at the beginning of a log. To make a new checkpoint, first write it into a temporary file, then rename the file in one shot so as to atomically switch from the old log to the new one. (See rename(2) in the 4.2bsd UNIX manual).

1. Synopsis

An overview of messages as they are used in ISIS.

2. Include file

```
#include <isis/cl.h>
```

3. Discussion

In ISIS, communication is via messages. Basically, a message is a container for some amount of data, organized as a vector of "fields", and possessing certain standard attributes. Specifically, each message has a *sender*, which is the address of the process that sent it (see ADDRESSING(TK)), a list of *destinations*, which is a null-terminated address list, and a set of fields containing data. The message editing system (see MSG_EDIT(TK)) provides a variety of routines to create and manipulate messages. Here, we confine ourselves to a summary that focuses on concepts rather than detail.

The type of a message is "message", and is predefined in cl.h. The data structure used is quite complex. An empty message is created by the routine msg_newmsg(), which returns a pointer to the message but doesn't fill out any of its fields. The sender field of a message is automatically set by the system in most cases, so applications can assume that this information is "secure". The routine msg_getsender(msg) returns a pointer to this field or ((address*) 0) if it is undefined. Similarly, the routine msg_setdests(msg,alist) sets the destination list of a message and msg_getdests(msg) returns a pointer to it. These can often be left undefined, for example if the message is to be broadcast: in such situations, the alist is computed by the ISIS communication subsystem and filled in automatically. (If the sender field is not filled in, the message subsystem uses the address of the process that called msg_newmsg() by default). The routine msg_getdests(mp,len), however, is quite useful: it returns a list of the destinations to which copies of a message were to be delivered. For example, this would be a good value to provide to coord-cohort() as an alist. If you use the destination list for any other purpose, be aware that your process group view should be checked to confirm that these members are still operational, see pg_getview() in PGROUP(TK) for details.

A message has no upper size limit, but ISIS tends to get sick when messages exceed a few hundred-thousand bytes in length. Some parts of the system break messages into 4k chunks for transmission, so 4k is a good upper limit. Since messages have overhead, the user-space available in a 4k message is only about 3.9k bytes.

Message *fields* are used to store data in a message. A field has a name, represented by an integer in the range 1-127, a type, a value, and a length. There are currently three types of fields, although more may be supported in the future:

1. Character fields are sequences of bytes having the designated length. These are not interpreted by the message subsystem. Users who employ a stub generator such as the UNIX XDR mechanism should think of the XDR output as a character field even if it represents multiple arguments or data items. The type of a character field is FTYPE_CHARS.
2. Long integers have different byte orders on different machines. Within ISIS, many data structures consist of vectors of long integers. A field containing long-integers will automatically be byte swapped on arrival at a remote machine if necessary, but the message editing system must be told that the field is not just a field full of characters. To do this, the type field should be specified as FTYPE_LONGINTS. The length of the field should be given in bytes. The message editing subsystem knows about byte orders automatically and will swap bytes as needed.
3. Other messages. The idea is that a message can be stuffed into another message, which is convenient when multiple messages need to be piggybacked to a single destination, or when extra fields need to be added to a message without risk of banging into fields already in the

message. The type of a message field is automatically set to FTYPE_MSG.

4. Note: ISIS currently uses an address format that is machine independent. Lest this change, a field type FTYPE_ADDRS is supported.

5. The following addition field types are support: FTYPE_SHORTINTS, FTYPE_SIDS (site-id's), FTYPE_PGVIEW (process group view).

The routines used to manipulate message fields are as follows:

msg_addfield(mp,fname,ptr,ftype,len)
: Adds a field named *fname* to the message having a value copied from the place the pointer points to and length given by len. The field name need not be unique. The type is as described above.

msg_getfield(mp,fname,inst,lenptr)
: The message is searched for the inst'th instance of the designated field name (the first instance is number 1). If found, a pointer to the value is returned and if lenptr is non-zero, the integer variable it points to is set to the length in bytes (or elements) of the character field (or long integer field).

msg_deletefield(mp,fname,inst)
: Deletes the i'th instance of the named field.

msg_addmsg(mp,fname,ptr)
: If ptr is a pointer to a message, creates a field with the given name containing this message.

msg_getmsg(mp,fname,inst)
: Returns a copy of the message in the inst'th field having the given fname.

There are additional routines that can be used to retrieve multiple values or messages at once when a message is expected to have several field instances with a single field name. Also useful is a procedure called msg_create() that creates a message and initializes a number of fields at the same time; it takes a list of field names, values, and lengths terminated by a field name of 0. See the MSG_EDIT(TK) documentation for details.

4. Field names

Some field names are standard in ISIS. These have 128-255 and are defined n msg.h, they can be fetched using msg_getfield but not set. Field numbers 1-127 are available for general use, and different applications will tend to use the same numbers for different purposes. Number 0 is not used. It is generally a good idea not to reuse field numbers within any single application; this avoids confusion.

5. Sending a message

A number of routines facilitate the transmission of messages. The most commonly used routines are the BCAST() routines; see BCAST(TK). In ISIS clients, the procedure isis_send(dest,msg) can be used to transmit a message to a destination given by the address *dest*. This routine is used heavily within the system. A second routine, isis_rpc(dest,msg), sends the message, waits for a reply, and then returns the reply message to the caller. The answer itself will be in the field FLD_ANSW. If the answer was given as a null pointer, this field will not be defined. If no reply will arrive because of a failure, isis_rpc returns a null pointer. Broadcasts are sent using any of a variety of procedures documented in BCASTS(TK).

6. Deleting a message; multiple copies of a message

Each message has a reference count which can be incremented by the procedure msg_increfcount(). The reference count is initially 0 and need only be incremented when the message is placed on a queue or otherwise passed to some task other than the original creator. Later, the message reference count is decremented by calling msg_delete(). If the count was 0, msg_delete() will free the message for reuse. While a message has a non-zero reference count, it is illegal to add or delete fields: such a message is said to be immutable, and optimizations taking advantage of this property have been included into the msg_addmsg() and msg_getmsg() utilities.

7. Byte swapping and addresses

The address format used in ISIS has been designed so that even in future versions of the system, there will be no need to byte-swap addresses when they are passed from process to process across machine boundaries.

8. Forwarding a message

Some systems allow a process to forward a message transparently: to the end recipient it looks as if this message came directly from the recipient. ISIS doesn't allow you to do this, although it would be easy enough to support if desired. The problem is that such a mechanism appears to break our security model. To forward a message using the current system, you would need to pack it into some other message using msg_addmsg() and then convince the destination to unpack the message and deliver it locally, by calling cl_local_delivery() on it.

1. **Message editing routines and their arguments**

```
/* msg_addfield : Add a new field to a given message. Only non-negative */
/*              field names are accepted.  Returns a pointer to the   */
/*              field.                                                */


char *
msg_addfield (msg, field, data, type, len)
 message   *msg;
 int       field, type, len;
 char      *data;


/* msg_addmsg  : Insert a msg2 into msg1 as a field          */
/*             Only non-negative field names are accepted.  */
/*             Returns pointer to field                     */

char *
msg_addmsg (msg1, msg2, field)
 message   *msg1, *msg2;
 int       field;


/* msg_copy: Make a copy of the given message */

message *
msg_copy (msg)
 message   *msg;


/* msg_delete: If the reference count of the given message is zero,  */
/*            release the space allocated to it.  Else, decrement   */
/*            the reference count. Called by the routine that       */
/*            allocated the message and by any routine that called  */
/*            msg_increfcount on the message                        */

void
msg_delete (msg)
 message   *msg;


/* msg_deletefield : Delete the given instance of a field */

void
msg_deletefield (msg, field, inst)
 message   *msg;
 int       field, inst;


/* msg_genmsg: Generate a message containing the fields passed as   */
/*           arguments, which are of the form field_name,          */
/*           pointer_to_data, field_length, ... , followed by 0  */
```

```
message *
msg_genmsg (field1, data1, type1, len1, field2, data2, type2, len2, ..., 0)
 int      fieldx, typex, lenx;
 char     *datax;


/* msg_getdests: Return a pointer to a null-terminated list of the       */
/*             destinations of a given message (null pointer, if none) */
/*             If a non-zero argument is given for n_dests, the number */
/*             destinations is returned.                                */

address *
msg_getdests (msg, n_dests)
 message   *msg;
 int       *n_dests;


/* msg_getfield: Return a pointer to the data in a given instance of a   */
/*             field and the length of the field if the last argument  */
/*             is non-zero.                                             */

char *
msg_getfield (msg, field, inst, len)
 message   *msg;
 int       field, inst;
 int       *len;


/* msg_getfields: Return an array of pointers to the data of up to the first */
/*             n instances of a given field, and their lengths (if the   */
/*             argument is non-zero). Return value: number of fields     */
/*             actually found.                                            */

msg_getfields (msg, field, pointers, lengths, n)
 message   *msg;
 int       field, lengths[], n;
 char      *pointers[];


/* msg_getiovec : Return a pointer to an array of iovec structures for */
/*             of a given message. See write(2) and read (2).       */
/* msg_getiovlen: Return the length of the iovec array.              */
/*    (These should really be macros).                                */

struct iovec *
msg_getiovec (msg)
 message   *msg;

msg_getiovlen (msg)
 message   *msg;


/* msg_getlen : Return the length of the transmittable part of a given message */
```

```
msg_getlen (msg)
 message    *msg;


/* msg_getmsg : Create a message from the contents of a given instance */
/*            of a field. (It is assumed that the field has a proper */
/*            message header.)                              */

message *
msg_getmsg (msg, field, inst)
 message    *msg;
 int        field, inst;


/* msg_getmsgs : Return a vector of messages created from up to the first */
/*               n instances of a given field, and return the actual    */
/*               number of such messages.  (It is assumed that each      */
/*               field has a proper header.)                       */

msg_getmsgs (msg, field, messages, n)
 message    *msg, (*messages)[];
 int        field, n;


/* msg_getreplyto : Return the address to send the reply for a given message */
/*                  Caution: this may not be the sender!                */

address *
msg_getreplyto (msg)
 message    *msg;


/* msg_getsender  : Return the address of the sender for a given message      */
/*                  Caution: this may not be the place to send replies!     */

address *
msg_getreplyto (msg)
 message    *msg;


/* msg_increfcount : Increment the reference count of a given message */
/*                   Caution: A message with multiple refs cannot be changed */

void
msg_increfcount (msg)
 message    *msg;


/* msg_newmsg: Create an new message with sender field filled in */

message *
msg_newmsg ()
```

```
/* msg_read: Read and return a message from a file descriptor */
/*          Second argument gives length if known, 0 otherwise */

message *
msg_read (sd, len)
int     sd;
int     len;

/* msg_reconstruct: reconstruct the argument into a message and return a ptr */

message *
msg_reconstruct (ptr)
char    *ptr;

/* msg_reconstruct_inplace: reconstruct the in place and return a ptr to its header */
/*          Caution: assumes the ptr points to a malloc region; will be freed    */
/*          automatically later by the message editing system                   */

message *
msg_reconstruct_inplace (ptr)
char    *ptr;

/* msg_setdest :  Set the destination field to the given destination */

void
msg_setdest (msg, dest)
 message    *msg;
 address    *dest;


/* msg_setdests :  Set the destination field to the given null-terminated */
/*              list of destinations                                      */

void
msg_setdests (msg, dests)
 message    *msg;
 address    dests[];


/* msg_setreplyto: Set the reply-to field to the given address          */
/*          Note:  Users cannot set the sender address -- this is automatic */

void
msg_setreplyto (msg,who))
 message    *msg;
 address    who;


/* msg_write: Write the given message on the given file descriptor */

msg_write (sd, msg)
 int     sd;
 message *msg;
```

1. Synopsis

The News Service allows an isis client to *post* messages which will automatically be forwarded to other processes that are *subscribers* of the news service.

2. Interface

```
    #include <isis/cl.h>

        news_post(slist, subject, mp, back)

        news_posta(slist, subject, mp, back)
          site_id  slist[];
          char     *subject;
          message  *mp;
          int      back;

        news_clear(slist, subject)

        news_clear_all(slist, subject)
          site_id  slist[];
          char     *subject;


        nback = news_subscribe(subject, entry, back);
          int  nback;
          char *subject;
          int  entry, back;

        news_cancel(subject);
          char *subject;
```

3. Discussion

To post a message, a process calls *news_post*. *Slist* is a list of sites to which the message will be forwarded; if a null pointer is given instead, the message will be forwarded to all operational sites. *Subject* is an arbitrary string of up to SUBJLEN characters. For every subject the news service at each site keeps a list of recently posted messages which new subscribers may look at. When a message is posted, the parameter *back* determines how long the message will be held as a "back issue". If back = 0, the message will be forwarded only to current subscribers and will be deleted immediately afterwards. If back is greater than zero, for example back = 5, the message will be held until five new messages have been posted to the same subject.

News_post uses CBCAST to broadcast the message to the news services at other sites. If it is important that all subscribers receive news messages in the same order, then *news_posta* should be called, which will use an ABCAST to post the message.

Messages kept as back issues on a certain subject may be deleted explicitly by calling *news_clear* (deletes all messages posted by the caller), or *news_clear_all* (deletes all messages posted by anybody).

A process that wishes to subscribe to a news subject calls *news_subscribe*, specifying an entry point (declared by isis_entry(), see ENTRIES(TK)) to which the news service will send messages posted to that subject. The parameter *back* specifies how many back issues the subscriber wants to receive. News_subscribe returns the actual number of back issues available, that will be sent to the subscriber.

When a message is posted, the news service automatically adds the two fields FLD_SUBJ and FLD_BACK containing the 'subject' and 'back' parameter from the news_post() call. A subscriber may inspect these fields simply by calling msg_getfield().

News_cancel cancels a subscription for a given subject.

4. Diagnostics

All routines return a nonzero value in the case of an error, except for news_subscribe, which indicates an error by returning a negative value. Note that news_post and news_posta do not wait for replies when broadcasting a message. Therefore a successful return does not yet guarantee that the message has been delivered to remote sites.

5. Bugs

A site crash wipes out all back issues held by the local news service. The news service does not save messages on stable storage, nor does it attempt to get back issues from some other site after a recovery.

This version of the news service does not provide any form of security. Any isis client can post and receive messages on any subject; it can delete any back issues it wants to (news_clear_all).

News uses a linear search to find subjects in its tables. Hashing should be used instead.

If the news service turns out to be heavily used, it might make sense to move it into the protocols process.

1. Synopsis

A package of routines implementing process groups and group addressing.

2. Interface

```
#include <isis/cl.h>

        isis_init(0);
        pg_init();

        . . .

        address pg_create(name,incarn)
          char *name;
          int incarn;

        address pg_lookup(name)
          char *name;

        pg_addmemb(gid, who)
          address gid, who;

        pg_leave(gid)
          address gid;

        pg_migrate(gid, newpname)
          address gid, newpname;

        pg_delete(gid)
          address gid;

        pg_addclient(gid, client)
          address gid, client;

        pg_delclient(gid, client)
          address gid, client;

        address *pg_getview(gid)
          address gid;

        pg_lockview(gid)
          address gid;

        pg_signal(gid, signo)
          address gid;
          int signo;

        pg_monitor(gid, routine, arg)
          address gid;
          int (*routine)();
          char *arg;

        pg_monitor_cancel(gid, routine, arg)
```

```
  address gid;
  int (*routine)();
  char *arg;

pg_join(gid, mp)
  address gid;
  message *mp;

pg_join_verifier(routine)
  int (*routine)();

pg_dump()
```

3. Discussion

This package maintains process group membership information. There are two ways a process can relate to a group:

1. As a *client* that sends requests to the group. Due to a restriction on addressing modes, we distinguish between the case of a client known to the group, with unrestricted permission to use the group address in its address lists (see BCAST(TK)), and the case of a client that the group doesn't know about in this sense, who can only broadcast to the group in a restricted manner (see BCAST(TK) again). Groups can promote a client to the more powerful form of addressing using pg_addclient(), but this must be done by a current member. The routine is idempotent, so calling it a few times won't crash the system or anything, but it might be slow. A client can monitor group membership changes, but will not receive broadcasts sent to the group and cannot initiate membership changes or add other clients on its own.
2. As a *member*. Group members receive messages sent to the group and have unlimited freedom to call the routines defined above. Group members implicitly have unlimited addressing freedom with respect to their group.

We now describe the various routines available to group members and their clients. See also the discussion in PROTECTION(TK), where the mechanisms for preventing unauthorized use of a group are documented.

a) pg_create() creates a new process group; its only member will be the caller process. The group may be given a symbolic name (if none is desired, a null pointer should be passed for the name). The system will not verify that the name is unique. It may also be assigned an incarnation number; this is done by and used by the recovery manager (see RMGR(TK) for details; this may be specified as 0 if the rmgr is not being used). The group will continue to exist until deleted with pg_delete unless all its members fail, at which time it will be deleted automatically. See BCAST(TK) for details concerning group addressing. A process can be a member of an unlimited number of groups.

b) pg_lookup(name) looks for a group with the given symbolic name and returns its group id. The search is done in all sites that are "local" and "long distance", but not those that are "remote" relative to the caller (see ADDRESSING(TK) for definitions of local and remote). If the name is not found, NULLADDRESS is returned. In the future, pg_lookup() will be extended to support some form of pattern matching and a permission scheme under which it will only be possible to lookup a name if one has "permission" to access it. If several groups match, the first address found is returned to the caller.

c) pg_addmemb() adds a new process to a preexisting group. It can only be done by a group member; this is to allow the group members to validate new potential members. It fails, returning an error code, if the group does not exist, the caller is not a member, or the process is already a member. Since this call can only be done by a member, pg_join() is provided as a convenient way to ask a group to let a potential member join.

d) pg_leave() deletes the caller from the designated group.

e) pg_migrate() simultaneously adds process "newpname" and deletes the caller process from the group. The discussion of pg_addmemb applies. It fails if the group does not exist, the caller is not a member, or newpname is not specified correctly. *This routine has not yet been implemented.*

f) pg_delete() deletes the designated group even if it still has members. The caller must be a member of the group. It fails if the group cannot be found or the caller is not a member.

g) pg_addclient() makes the group directly accessible by the designated client, but without making the client a member of the group. This is necessary if the client is to use some of the more sophisticated addressing modes identified in the BCAST(TK) routines. Should the client fail, it will automatically be deleted from this list.

h) pg_delclient() deletes a direct access client.

i) pg_getview() returns the current membership of a group as a pgroup data structure, defined in pr_group.h and automatically included by cl.h. The caller need not be a member and the group need not be directly accessible. The view is not guaranteed to remain unchanged. It is, however, guaranteed to have the same value when different recipients of a message all call pg_getview() when the message first arrives (but without doing a t_wait() first). A null pointer is returned if the group is not found.

j) pg_lockview() locks the current view against changes and returns the view, which the caller can use to compute an action that depends on the current membership. The lock is automatically released the next time a broadcast is done to the group by this process, or if the process fails. It returns a null pointer if the group is not found. *This routine is not yet implemented.*

k) pg_signal() sends a UNIX signal to the members of the designated groups and processes, which are identified by a null-terminated address list. It fails if the caller is not a member of the group or the group does not exist.

l) pg_monitor() monitors the designated group for membership changes. The caller must be a member, the request fails if this is not the case. Should the membership change, the callback routine is invoked as: routine(pg, arg); where pg is a pointer to a pgroup data structure containing the new view and the argument is the one given in the monitor request.

m) pg_monitor_cancel() cancels a pg_monitor request. The arguments must match those for the pg_monitor. It fails if the monitor request is unknown.

n) pg_join() is a toolkit routine that does an RPC to the designated group, passing the designated message. The message pointer can be null if the group doesn't do verification. The message is delivered to the join verification routine (see below). If this routine is undefined or returns 0, the join is permitted. If it is defined and returns an error code (a negative number), the join is aborted and pg_join() returns this error code. If the caller is added to the group successfully, pg_join() returns 0. A process can be a member of an unlimited number of groups.

m) pg_join_verifier() is a toolkit routine with which the members of a group can specify a routine that will validate new join requests. The routine is later invoked as

```
routine(mp)
  register *mp;
  {
  }
```

where mp is the message sent in the pg_join() request. The id of the group being joined is available in the system field SYSFLD_GID of this message. The join verifier is only called at one of the members of the group, but the actual member that will be asked to do the verification may vary. All members should therefore define this routine if any does and all should use the same verification rule.

Except when otherwise indicated, all the routines return 0 in the event of normal termination and -1 if an error occurs.

1. Synopsis

Some thoughts on protection in ISIS.

2. Relevant Interfaces

```
pg_join(...)
pg_migrate(...)
pg_signal(...)
au_verify(...)
au_permit(...)
au_revoke_perm(...)
bcast(...)
```

3. Discussion

Because ISIS will operate in large networks with multiple protection domains present, protection within ISIS itself poses difficult design issues. Two basic approaches have been considered in this connection.

The first approach is to encrypt capabilities (group and process id's) and use the encrypted capabilities to mediate access to groups. This approach has been rejected because the size of the encrypted capability would have to be very large, hence address lists (which are common in ISIS) could get very big.

The alternative, which we implemented, treats authentication as an application-level problem, but provides a reasonable degree of support for authenticating access. The approach is as follows.

First, members of a process group are given a chance to check the credentials of a process wishing to join the group. See pg_join() for details. The idea is that the group members specify a join_verifier routine and it checks the legality of the join request. ISIS provides the sender's address in a *secure form*, but leaves it to the recipient processes -- the group -- to check the sender's user identification. This is because many operating systems simply do not provide ISIS with a mechanism for securely deducing any more information than this.

A reasonably secure mechanism, if security is your goal, would be to use public key encryption as part of these authentication procedure. If a client wishes to join, it would be required to present credentials to the group, obtaining these from a file that only it can access, and encrypting the information to prevent eavesdroppers from learning anything useful.

We also support a mechanism for authentication of individual requests. See AUTHEN(TK) for details. The interface is very similar: the group specifies a routine that authenticates individual messages, although in this case (to avoid extra work), it is also possible to permit all messages from a particular caller to be processed or to revoke a previously granted permission. The mechanism uses the *filter* concept outlined in FILTER(TK).

1. Synopsis

A simple program demonstrating the use of the recovery manager routines.

2. Program Source

```
/*****************************************************************************
 *
 * rdemo.c -- a simple demonstration on how to use the rmgr routines
 *
 * What this program does:
 *   The program creates a process group named "recovery-demo" (or joins this
 *   group if it exists), prints the message "rdemo: startup complete, waiting
 *   for command messages", and (you guessed it) waits for 'command' messages
 *   to arrive from some mysterious source. The purpose of this program is to
 *   demonstrate how to use the recovery manager routines to have the program
 *   restarted automatically and to create or join a process group after a crash.
 *
 * How to run this program:
 *
 *  1. Start up isis on all sites.
 *  2. Install an entry in the recovery manager restart database by typing:
 *        rmupdate 1 rdemo-program rdemo rdemo 1252
 *     This command creates an entry in the restart database for site 1, with
 *     key = "rdemo-program", program = rdemo, and argv = {rdemo 1252}.
 *     The rdemo program expects argv[1] to contain the internet port number
 *     for talking to isis. Replace 1252 by the correct number for that site. See
 *     INIT(TK), FILES(TK), RMUPDATE(TK) for details. You have to repeat
 *     this command for every isis site on which you want to run the demo.
 *  3. Start the demo program at one of the sites (e.g. site 1) by typing
 *        rdemo 1252
 *  4. Start the demo program at the other sites.
 *  5. Now you can experiment to see what happens if you crash one of the
 *     programs (with cntrl-C or 'kill') or one of the sites (by killing
 *     the isis protocols process).
 *
 *  The program should be restarted automatically each time it is killed,
 *  or after a site recovers from a crash. If you crash all sites and then
 *  bring them up again, the process group will be recreated on the site
 *  that died last. The rdemo programs at the other sites will wait until
 *  rdemo is restarted on the 'last surviver' site.
 *
 *  To remove the rdemo entry from the restart database type
 *     rmupdate 1 rdemo-program
 *  at site 1, and similar for the other sites.
 *
 *****************************************************************************/

#include <stdio.h>
#include "cl.h"
#include "cl_rmgr.h"

#define MSG_COMMAND  (USER_BASE+0)      /* isis entry number for command message */
#define FLD_TEXT     1                  /* message field for command text */
```

```
/*********************************************************************
 *
 *  main_task -- starts up the process group
 *
 *********************************************************************/

void main_task()
{
    address   gid;            /* process group address */

    /*
     * Register this process with the recovery manager.
     * This is only necessary if the program was not started by the recovery
     * manager (i.e. the first time, when it is started "by hand"). However,
     * it does not hurt to always call rmgr_register().
     */
    rmgr_register("rdemo-program");
    /*
     * Restart the process group.
     * Rmgr_restart() will call rmgr_getinfo() and based on that information
     * decide whether to just join the group, create the group, wait for another
     * site to create it, .... See RMGR(TK) for details.
     */
    gid = rmgr_restart("recovery-demo");
    if (gid.site == 0) {
        printf("rdemo: rmgr_restart failed\n");
        exit(-1);
    }
    /*
     * Start recovery manager view logging.
     * After this call the latest pgroup view is automatically
     * save on stable storage.
     */
    rmgr_start_log(gid);

    printf("rdemo: startup complete, waiting for command messages\n");
}


/*********************************************************************
 *
 * msg_command -- message handler for command messages
 *
 *********************************************************************/

void msg_command(mp)
    message *mp;
{
    char    *text;

    text = msg_getfield(mp, FLD_TEXT, 1, (int *)0);

    if (strcmp(text, "quit") == 0) {
        /*
         * Call rmgr_unregister to tell the recovery manager that
         * the program wants to exit without being restarted.
         */
        rmgr_unregister();
        exit(0);

    } else if (strcmp(text, "crash") == 0) {
        /*
         * Commit suicide. The recovery manager will notice that the
         * program has died and will restart it automatically.
         */
        exit(0);

    } else if (strncmp(text, "echo", 4) == 0) {
        /*
         * Echo the message text to the screen.
         */
        printf("rdemo: %s\n", text+4);

    } else {
        printf("rdemo:  unknown command: %s\n", text);
    }
}


/*******************************************************************
 *
 * main
 *
 *******************************************************************/

main(argc, argv)
    int  argc;
    char *argv[];
{
    int  client_port;  /* port number for talking to isis */

    /*
     * get client_port from argv
     */
    if (argc != 2 || (client_port = atoi(argv[1])) == 0) {
        fprintf(stderr, "usage:  %s port\n", argv[0]);
        exit(-1);
    }
    /*
     * set up isis stuff
     */
    isis_init(client_port);
    rmgr_init();
    isis_entry(MSG_COMMAND, msg_command, "msg_command");
    /*
     * fork main task
     */
    t_fork_delayed(main_task, 0, 0);

    for (;;) {
        run_tasks();
        isis_read();
    }
}
```

1. Synopsis

A toolkit routine for managing replicated data.

2. Interface

```
    #include <isis/cl.h>

    In the service:
          isis_init(0);
          r_init(item_name, opener, sizer, reader, writer, fsyncer, default)
            char *item_name;
            int (*opener)(), (*sizer)(), (*reader)(), (*writer)();
            int (*fsyncer)(), (*default)();

    In the client:
          isis_init(0);
          . . .
          fd = r_open(alist, item_name, how, fmode)
            char *item_name;

          r_lseek(fd, offset, mode)

          nbytes = r_read(fd, buffer, len)
            char *buffer;

          nbytes = r_write(fd, buffer, type, len)
            char *buffer;

          nbytes = r_xwrite(fd, buffer, type, len)
            char *buffer;

          r_fsync(fd);

          r_close(fd);
```

3. Discussion

A simple interface to replicated data is presented. The interface largely emulates the existing UNIX file system interface, although replicated data need not be stored on disk. Basically, the managers of the replicated data item specify the routines that will read, write, etc. and the type of broadcast to use by "default" when updating this data; in the xwrite() ("exclusive mode write") case CBCAST is always used regardless of the default. Normally, the default routine will either be CBCAST or ABCAST (see VSYNC(TK)). CBCAST would be used for structures that are insensitive to the order in which updates are done to different data items; ABCAST when a structure might behave differently for different update orders. The type field should be set as for a msg_addfield (see MESSAGES(TK)).

In the service, the routines are invoked as follows:

opener(item_name,how,fmode)

The service should "open" the designated item_name, the arguments can be defined by the service but are intended to mimic the argument to the unix open system call. The opener routine should return -1 on an error, setting the global variable "r_errno" to the error code, and it should return 0 if the open succeeded. No state is saved on an open. The designer should be aware that the file descriptor returned to the client will not be determined from the code returned by the open routine. The client file descriptor is allocated from a per-client table of open "files", corresponding to the various successful opens that have been done. In fact, it is not required that the number returned by different representatives of the service be the same. If some representatives return error codes, the request is assumed to have failed and one of those codes is returned to the caller. If all return success, the request succeeds. The designer of the system may assume that the same sequence of r_open requests is seen by all members of the service, but that the sequence of r_close invocations may differ from representative to representative. The designer should arrange that the representatives are left in equivalent states after the open call returns, which normally means that all representatives should do the same thing on an open.

sizer(item)
: The service should return the size of the file. The return code and error number are as above.

reader(item,offset,buffer,len)
: The service should do a read at the designated offset. The number of bytes read should be returned, or -1 if an error occurred with the error code stored in r_errno.

writer(item,offset,buffer,len)
: The service should do a write at the designated offset. The number of bytes written should be returned, or -1 if an error occurred with the error code stored in r_errno.

fsyncer(item)
: The service should do the equivalent of an fsync(), returning only when the outputs previously done on the item have completed.

4. What about recovery from failures?

After a failure, a member of the recovery service may want to rejoin the service. It should do this using the state transfer tool described in STATE_XFER(TK). To have a restart initiated automatically, use the recovery tool described in RMGR(TK). In the case of a recovery from total failure, the service should take the following actions to recover its state: Save checkpoints of the replicated data periodically in a file, using fsync to verify that it has been fully flushed to disk. It is best to write a temp file and then rename it to avoid the risk that a crash will leave you with a partially updated checkpoint. Or, you could update a stable copy of the file on every write. Using the recovery manager, you can determine whether to rejoin the group (if it survived you or someone else recovered first) or to reload your checkpoint.

5. Transactions on replicated files

You can use this tool in conjunction with the ISIS transaction facility to obtain transactions on replicated files.

1. Synopsis

A package of routines for remote execution of a program.

2. Interface

```
    #include <isis/cl.h>

        isis_init(0);

        . . .

        r_exec(sites, prog, args, env, user, passwd, alist)
          site_id *sites;
          char *prog, **args, **env, *user, *passwd;
          address *alist;
```

3. Discussion

This toolkit routine is provided as an alternative to using the UNIX "rexec" facility. The specified program is executed at each specified site. The file descriptors that are initially open (stdin, stdout and stderr) point to the system console of the machine on which the rexec is done, so be careful what you print. You should use the normal UNIX facility if you prefer for these to point to the console of the machine where the program was run from.

Unfortunately, the UNIX architecture makes it impractical to return an definite indication of whether the exec actually succeeded. However, for the cases where it was apparently possible to do the rexec and it was attempted, the alist will contain a null-terminated list of process addresses corresponding to the processes that were created to do this. For cases where the rexec could not be done but some detectable error occurred the alist entry for the corresponding site/incarn will have process-id number 0 and the entry field will be equal to the value of the UNIX errno variable at that site. If a site/incarn is not operational, no alist entry will be made. The number of entries in the alist is returned, or an error code if the CBCAST to the rexec processes failed for some reason.

Rexec encrypts its message to prevent unauthorized access to the user name / password. Remotely, it encrypts the password and then compares this with the version in the local password file. This mechanism is a bit awkward, but it does provide at least a modicum of security. The remote program will be executed in the home directory of the designated user.

4. BUGS

It is unfortunate that rexec() cannot indicate whether the exec actually succeeded. The user-id and password are currently ignored. Everything runs under the isis account.

1. Synopsis

A toolkit routine for assisting in recovery from failures. This tool is still under design. The current version of the recovery manager consists two independent parts: *Automatic Process Restarting*, and *Process Group Logging/Restarting*.

2. Interface: Automatic Process Restart

```
#include <isis/cl.h>

        rmgr_update(key, program, argv, envp);
          char *key;
          char *program;
          char *argv[], *envp[];


        isis_init(0);

        ...

        rmgr_register(key);
          char *key;

        rmgr_unregister();
```

3. Discussion

The recovery manager keeps a database of programs that it will restart after a site recovers from a crash. The function *rmgr_update* atomically updates the restart database. *Key* is an arbitrary string of up to RMLEN characters that uniquely identifies an entry in the restart database. *Program* is a program name, and *argv* and *envp* are vectors of argument and environment strings as in execve (2). Rmgr_update searches the database for an existing entry with the given key. If such an entry is found, program, argv, and envp will be replaced by the new values; otherwise a new entry will be created. Rmgr_update may be used to delete an existing entry by specifying program as a null pointer. The utility program rmupdate (see RMUPDATE(TK)) provides a simple user interface for rmgr_update.

The recovery manager keeps "watching" all processes that it has started up. Should any of them abort, say due to a software error, it will automatically be restarted. A process that was *not* started by the recovery manager may add itself to the list of processes being watched, by calling *rmgr_register(key)*, where *key* must refer to an existing entry in the restart database. A process that wants to exit without being restarted has to call *rmgr_unregister* before exiting.

4. Diagnostics

All routines return a nonzero value in the case of an error. For rmgr_update the following two error codes are defined:

> RM_ELOCKED: The restart database is locked because another process is currently updating it. The call should be retried.
>
> RM_ENOTFOUND: Rmgr_update was called to delete an entry (program = NULL), but no entry with the given key exists in the restart database.

Rmgr_update returns a negative value if it fails for any other reason.

It is no error to call rmgr_register if the process is already registered. This may be used to change the restart database entry associated with the process. Rmgr_register does not check whether the given key exists in the restart database. The recovery manager will print a message on stderr if it is unable to restart a process because it cannot find the entry in the restart database.

5. Interface: Process Group Logging/Restarting

```
#include <isis/cl.h>
    isis_init(0);
    rmgr_init();
    ...

    rmgr_start_log(gid);
      address gid;

    rmgr_stop_log(gid);
      address gid;

    rmgr_info  *rmgr_getinfo(pgname, noblock);
      char *pgname;
      int  noblock;


    gid = rmgr_create(rmi)
      rmgr_info  *rmi;

    gid = rmgr_join(rmi, mp)
      rmgr_info  *rmi;
      message   *mp;

    gid = rmgr_restart(pgname)
      char *pgname;
```

Defined in isis/cl_rmgr.h (included in isis/cl.h):

```
    typedef struct {
       int     rm_mode;
       pgroup  rm_view;
    } rmgr_info;

    #define RM_LOG    0x01
    #define RM_RECENT 0x02
    #define RM_SURE   0x04
```

6. Discussion

The recovery manager also assists in recreating a process group after all its members have crashed. When restarting a process group after a total crash, it is desirable to find out out which process was the last one to fail. For this purpose a log of changes in the process group view is kept on stable storage.

(At least) one member of a pgroup at each site should call *rmgr_start_log* (the process has to be a member of the group). This call saves the current pgroup view in a file on disk, and arranges for the file to be updated whenever the view changes. The function *rmgr_stop_log* disables automatic updates to the view file.

When a program is restarted after a site recovers from a crash, the program can call *rmgr_getinfo* to get information about the state of a process group before the crash. Rmgr_getinfo reads the last pgroup view that was stored on disk into the field *rm_view* and sets some flags in *rm_mode*, which are to be interpreted as follows:

1. RM_LOG not set: A pgroup view file does not exist or it is empty. Interpretation: This is the first time that a process at this site becomes a member of the group.
2. RM_LOG, RM_SURE set, RM_RECENT not set: A view has been read from disk; however, this view is not the most recent one among the views stored in view files at other sites. Interpretation: A process (or processes) at this site was a member of the group when the site (or just that process(es)) crashed. Other members of the group at other sites were still alive after this crash.
3. RM_LOG, RM_RECENT, RM_SURE set: A view has been read from disk; no other site has a more recent view recorded on disk. Interpretation: All members of the group have crashed. This site was one of the last sites up before the crash occurred. Rm_view contains the list of last survivers before the crash.

In order to set RM_RECENT correctly, rmgr_getinfo may need to access view files at other sites. Therefore rmgr_getinfo might block, waiting for other sites to recover, before it can decide whether the local view file contains the most recent pgroup view. In particular, *if rmgr_getinfo returns with all flags set* (case 3.), *the programmer can assume that all sites mentioned in rm_view have recovered from the crash*. This fact may be used to start a coordinator-cohort style protocol among those sites for application specific recovery.

If this behavior is not desired, rmgr_getinfo should be called with a non-zero value for the parameter *noblock*. In this case rmgr_getinfo will not block, but it may return with RM_LOG, RM_RECENT set, and RM_SURE not set, indicating that none of the sites *that are currently up* has a more recent view stored on disk.

The interpretation given above is only valid, if the recovery manager routines are used according to the following rules:

1. A program should call rmgr_start_log as soon as it has joined the group and has completed local initialization actions. If the program is restarted after a crash, rmgr_start_log should be called after rejoining the group and performing local cleanup/recovery actions.
2. After a total pgroup crash one of the last survivers should create the group again by calling pg_create(pgname, rm_view.pg_incarn+1). It is important that the incarnation number of the new group is greater than the one recorded last on disk. Rmgr_start_log should be called after global cleanup/recovery actions have been completed.

The routines *rmgr_create* and *rmgr_join* may be used to restart a process group after a total crash. Rmgr_create creates a new incarnation of the process group (based on the rmgr_info obtained by rmgr_getinfo) and announces the new gid on the news (see NEWS(TK)). This routine must be called by one of the last survivers. Other recovering group members should call rmgr_join, which will wait for the news announcement, and will then join the new group incarnation. Rmgr_join checks to see if the group still/already exists before it blocks waiting for news announcements; so it can also be used to rejoin a group after a local crash. The parameter 'mp' is passed to pg_join (see PGROUP(TK)).

Rmgr_restart provides a simple, minimal interface to rmgr_getinfo, rmgr_create, and rmgr_join. It assumes that there is only one member of a process group at each site. The code for rmgr_restart is given below; it illustrates the use of rmgr_getinfo, rmgr_create, and rmgr_join:

```
    address rmgr_restart(pgname)
        char *pgname;
    {
        rmgr_info *rmi;
        int          create_flag;
        address      gid;
        message      *mp;

        /*
         * Find out whether to create or join the group
         */
        rmi = rmgr_getinfo(pgname, 0);

        if (! (rmi->rm_mode & RM_LOG)) {
              /*
               * No previous view logged at this site: assume this is the first time
               * the group is started up. Check if the group already exists. If not,
               * create the group.
               */
              gid = pg_lookup(pgname);
              create_flag = (gid.site == 0);

        } else if (rmi->rm_mode & RM_RECENT) {
              /*
               * This site has a copy of the most recent group view in its view log file:
               * assume that all group members have crashed and that this site was one
               * of the last survivers. If this site is the first one in the list of last
               * survivers then create the new group incarnation; otherwise wait for
               * somebody else to create the group, and then join it.
               */
              create_flag = (rmi->rm_view.pg_alist[0].site == my_site_no);

        } else {
              /*
               * The group view stored at this site is not the most recent one in the
               * system: assume that this is a recovery from a local crash. Simply
               * rejoin the group, but use rmgr_join in case the group has crashed
               * in between.
               */
              create_flag = 0;
        }

        /*** create or join group ***/
        if (create_flag) {
              return rmgr_create(rmi);
        } else {
              mp = msg_newmsg();
              gid = rmgr_join(rmi, mp);
              msg_delete(mp);
              return gid;
        }
  }
```

7. Diagnostics

In case of an error rmgr_start_log and rmgr_stop_log return a nonzero value, rmgr_getinfo returns a null pointer, and rmgr_create, rmgr_join, and rmgr_restart return NULLADDRESS.

8. Bugs

Entries in the restart database are stored in the following format:

    "key" program {arg1, arg2, ..., argn} {env1, env2, ..., envn}

The rmgr will not work properly if the key contains quote characters, or if one of the argument or environment parameters contains curly braces or commas.

A much fancier recovery manager will be introduced eventually. The functions of this one will be preserved, but perhaps not the interface it supports.

1. Name

rmupdate - update the restart database used by the recovery manager

2. Synopsis

```
rmupdate  site_no  key
rmupdate  site_no [-E] key program arg0 arg1 ...
```

3. Description

Rmupdate is a utility program for updating the restart database used by the recovery manager. It calls rmgr_update (see RMGR(TK)) with the arguments supplied on the command line. *Site_no* specifies on which site the database should be updated. It refers to the site-id of the site as it is found in the sites file (see ADDRESSING(TK), site table). The parameters *key, program, arg0, arg1,* ... are passed to rmgr_update without change. If the *-E* option is used, rmgr_update is called with the environment from which rmupdate was started; otherwise rmgr_update is called with an empty environment.

4. Examples

Assume that the restart database at site 2 does not yet contain an entry with the key "test1". The command

```
rmupdate  2  test1 /isis/test/testprog testprog 1461 -v
```

creates the following entry in the database:

```
"test1"  /isis/test/testprog  {testprog, 1461, -v} {}
```

If later the command

```
rmupdate  2  -E test1 /isis/test/testprog testprog 1464
```

is issued, the entry will be replaced by something like

```
"test1"  /isis/test/testprog  {testprog, 1464}
{HOME=/isis/schmuck, PATH=.:/usr/local:/usr/bin, TERM=vt100, USER=schmuck}
```

Finally, the command

```
rmupdate  2  test1
```

deletes the entry from the restart database.

5. Bugs

It is currently not possible to specify environment values explicitly.

1. Synopsis

A package of routines implementing distributed semaphores.

2. Interface

```
    #include <isis/cl.h>

        isis_init(0);

        . . .

        Pb(alist, sname, free_on_failure)
          address *alist;
          char *name;
          address free_on_failure;

        Pg(alist, sname, free_on_failure)
          address *alist;
          char *name;
          int free_on_failure;

        Vb(alist, sname)
          address *alist;
          char *name;

        Vg(alist, sname)
          address *alist;
          char *name;

          sema_xfer_out(addr, len)
            char **addr;
            int *len;

          sema_xfer_in(addr, len)
            char *addr;
            int len;

        sema_dump()
```

3. Discussion

The semaphore tool is used for synchronization in a process group setting. By employing it, a process can obtain mutual exclusion with respect to some set of other processes that know of the semaphores it is using. The argument "alist" is a null-terminated address list that identifies the processes and process groups where the semaphore lives. All semaphore routines return 0 in the normal case and -1 if the processes corresponding to the alist have all failed.

The tool provides both binary (true/false) and general (integer valued) semaphores. Each semaphores is identified by null terminated character string, which need not be declared prior to the first use. A general semaphore will block if the number of Vg() operations done since the semaphore was first referenced is smaller than the number of Pg() operations done so far, including the current one. A binary semaphore will block unless a Vb() was done subsequent to the last Pb(). That is, a general semaphore is initially 0 and a binary semaphore is initially false.

The semaphore scheme is a "fair" one: P() requests are satisfied in the order they are received.

The argument "free_on_failure" indicates how the semaphore subsystem should handle the failure of a process which has done a P() and has not yet done a matching V(). If this argument is null (actually, NULLADDRESS), the semaphore subsystem will not worry about failures. If the argument is a group id, the semaphore system will watch the caller by monitoring that group, to which the semaphore holders must also belong. If the holder fails, a V() of the appropriate type is performed automatically. Semaphore users must either employ this mechanism or some mechanism with equivalent functionality to avoid deadlocks when a semaphore holder fails. Use of an alternate mechanism might be more appropriate if some cleanup actions must be taken on behalf of the semaphore holder before the mutual exclusion it held can be released.

4. Comment

Semaphore synchronization is compatible with all tools that maintain replicated data.

5. State transfer

To generate a block containing the semaphore "state", call sema_xfer_out; it will assign values to addr and len as required by XFER(TK). The block can be read in using sema_xfer_in(). Only one block is needed for the semaphore state; the length will depend on the number of semaphores in active use.

6. Bugs and restrictions

To make use of the free on failure option, a semaphore operation must be applied to a process group to which the caller belongs. This is because free on failure uses the watch tool, and the watch tool currently only supports watching process groups to which one belongs. This restriction will eventually be eliminated.

1. Synopsis

Startup sequence. for processes using the TASK mechanism. Non-TASK use of ISIS is not yet supported, so you MUST use this interface.

2. Interface

```
#include "cl.h"

/* Entry code by which eat_msg() can be called */
#define EAT_MSG (USER_BASE+0)

main(argc, argv)
  char **argv;
  {
      int foreground(), eat_msg();

      /* Initialize connection to ISIS */
      isis_init(0);

      /* Initialize toolkits */
      ... toolkit init calls ...

      /* Initialize entry points this client will support */
      isis_entry(EAT_MSG, eat_msg, "eat_msg");

      /* Fork off the foreground task, if any */
      t_fork_delayed(foreground, 0, 0);

      /* Main loop: run tasks and receive messages */
      forever
      {
          run_tasks();
          isis_read();   /* This blocks, but see below */
      }
  }

/* This task is the "main" procedure of the program */
foreground()
  {
      ... stuff ...
  }

/* Routines to process received messages */
eat_msg(mp)
  message *mp;
  {
      ... stuff ...
  }

/* Soft recovery from an ISIS system crash that left me running */
isis_failed()
  {
      /* In fact, I prefer to print a message and die */
```

```
        return(-1);
    }
```

3. Discussion

The above program executes a typical st[illegible]up sequence by initializing a connection to ISIS, declaring the messages sent to entry number EAT_MSG will be handled by a procedure called eat_msg(), and then spawning a foreground task that acts as the 'main procedure' for the program. The actual main procedure then loops running tasks and reading messages; it may also want to do non-blocking IO on other I/O channels. Messages from the ISIS protocols process and other remote processes are received over the file descriptor called isis_socket (a global integer). The routine isis_read() will read a message over this socket, blocking until one is received. It then spawns a task to deal with the arriving message.

The reader should refer to INIT(TK) for information about the mysterious argument to isis_init().

To do non-blocking IO from ISIS it would be best to change the main loop to do a select. For example, the following code either reads from isis or from the file descriptor "spcl_fdes", depending on which one has data available.

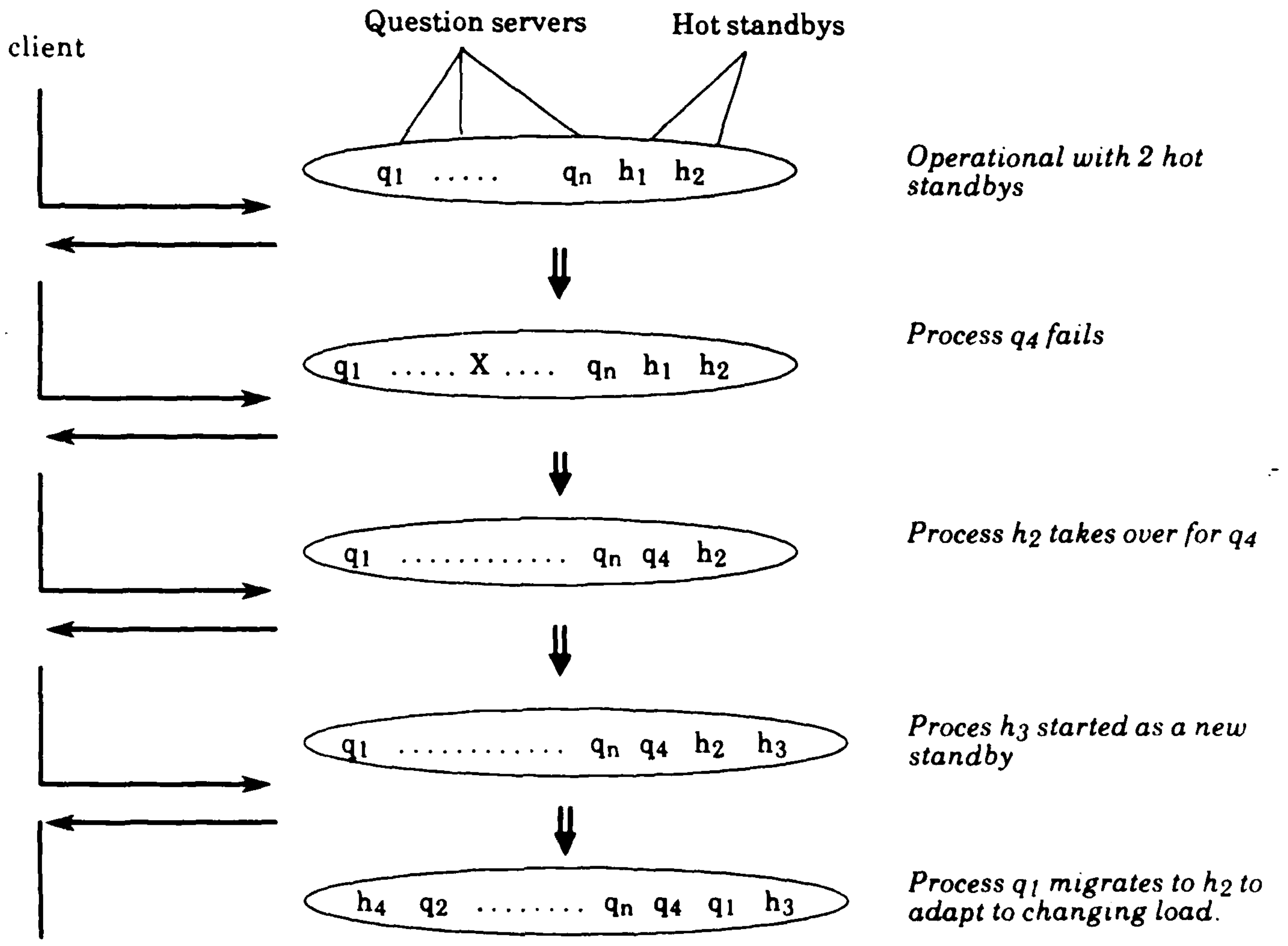
```
    #include <sys/types.h>
    #include <sys/time.h>

        ....

        forever
        {
            fd_set in_mask;
            extern isis_socket;
            run_tasks();
            FD_ZERO(&in_mask);
            FD_SET(isis_socket, &in_mask);
            FD_SET(spcl_fdes, &in_mask);
            /* Block until input is available */
            select(32, &in_mask, (fd_set*)0, (fd_set*)0, (struct timeout*)0);
            /* Read from ISIS and create associated task to run later */
            if(FD_ISSET(isis_socket, in_mask))
                isis_read();
            /* Read from special file descriptor */
            if(FD_ISSET(spcl_fdes, in_mask))
                spcl_read();
        }
```

The above code works as follows. Within a single address space, the foreground task and any active message-processing tasks will co-exist, switching off using a coroutine mechanism styled after monitors, as described in TASKS(TK). In this particular case, arriving messages with entry number EAT_MSG will be passed to the eat_me() routine, which should process the message and reply if necessary. Meanwhile, if data becomes available on spcl_fdes the routine spcl_read will be called; it should either do the read immediately or fork off a task to do it when run_tasks is next run (in fact, one might simply fork off the task directly, as in t_fork_delayed(spcl_read, spcl_fdes, 0)).

Unless some task explicitly calls "exit", this program will run until an error communicating with ISIS occurs or the site fails. In particular, if the foreground procedure returns, the program just becomes a passive service responding only to new requests. If the foreground procedure remains active, blocking periodically or doing calls to other processes, messages will be received while it is blocked. If it enters an infinite computational loop, it will not be interrupted. In addition, if it


client
Question servers
Hot standbys
q_1 ..... q_n h_1 h_2
Operational with 2 hot standbys
q_1 ..... X .... q_n h_1 h_2
Process q_4 fails
q_1 .... q_n q_4 h_2
Process h_2 takes over for q_4
q_1 .... q_n q_4 h_2 h_3
Proces h_3 started as a new standby
h_4 q_2 .... q_n q_4 q_1 h_3
Process q_1 migrates to h_2 to adapt to changing load.

Client uses group RPC to query a 20-question program while it dynamically reconfigures after failure reads from a blocking IO device like the keyboard, messages will NOT be accepted until the IO terminates.

A more useful foreground procedure would be the following: it creates a group:

```
foreground()
 {
     gid = pg_create("group_name");
     if(gid.site == 0)
         panic("create of group <group_name> failed!\n");
 }
```

A more complex mechanism might start multiple group members up using the isis remote exec facility (REXEC(TK)) and verify that each member is allowed to join the group. An example of this is given below.

Here is a second example: a process that wishes to act as a client to the example_group defined above.

```
foreground()
 {
     /* Example of process group creation and a broadcast */
     static address addrs[2];
     static char answ[8];
     message *mp;

     addrs[0] = pg_lookup("group_name");
     if(addrs[0].site == 0)
        panic("pg_lookup <group_name> failed");
     mp = msg_newmsg();
     addrs[0].entry = EAT_MSG;
     if(CBCAST(addrs, mp, ALL, answ, 8, (address*)0) != 1)
        panic("Got unexpected number of replies from CBCAST");
     printf("After CBCAST: received <%s>\n", answ);
     exit(0);
 }
```

It should be noted that this client doesn't obtain "direct access" to the group. To give it direct access it would be sufficient for the group member to call pg_addclient(sender) in the eat_msg() routine. The termination of the client would automatically trigger a pg_delclient().

The code below consists of a twenty-questions program and a question/answer program that acts as its client. The twenty questions program assumes that its remote representatives will start themselves up. More realistic would be to use the 'r_exec' facility for this purpose, but this would make the example a bit too complex.

Both programs reference the following include file:

```
#define TWENTY_QUERY    (USER_BASE+0)
#define TWENTY_DB_INIT  (USER_BASE+1)

#define TWENTY_CAT      0
#define TWENTY_CLASS    1
#define TWENTY_QUES     2
#define TWENTY_DB       3
```

The question database program is as follows:

```
/*
```

```
 *    A twenty questions program
 */

#include "cl.h"
#include "twenty.h"

int     main_proc(), join_proc(), twenty_db_init(), twenty_query();
address gid, atoaddr(), pg_lookup();

int     main_prog;
int     verbose;
int     my_number;
int     CLIENT_PORT;

main(argc, argv)
  char **argv;
  {
        while(argc-- > 1)
            switch(**++argv)
            {
              default:
              badarg:
                panic("Bad argument: <%s>\n", *argv);

              case '-':
                switch((*argv)[1])
                {
                  case 'm': ++main_prog; continue;
                  case 'v': ++verbose;   continue;
                  default:  goto badarg;
                }
              case '0': case '1': case '2': case '3': case '4':
              case '5': case '6': case '7': case '8': case '9':
                CLIENT_PORT = atoi(*argv);
                continue;
            }

        /* Connect to ISIS, then fork off appropriate procedure */
        isis_init(CLIENT_PORT);
        pg_init();

        isis_entry(TWENTY_DB_INIT, twenty_db_init, "twenty_db_init");
        isis_entry(TWENTY_QUERY, twenty_query, "twenty_query");


        if(main_prog)
            t_fork_delayed(main_proc, 0, 0);
        else
            t_fork_delayed(join_proc, 0, 0);

        /* Now enter ISIS main loop */
        forever
        {
            run_tasks();
            isis_read();
        }
    }

    condition      mcount;
    int            nmembers;

    /* Monitor changes to view */
    tmon(gid, pg, arg)
      address gid;
      pgroup *pg;
      char *arg;
     {
          if(pg->pg_nmemb == nmembers)
          {
             t_sig_delayed(&mcount, 0);
             pg_monitor_cancel(gid, tmon, 0);
          }

     }

    #define NMEMBERS       5

    #define NCAT          10
    #define NLINES       200
    #define NFIELDS       10
    #define STRLEN        10

    char   db[NLINES][NFIELDS][STRLEN];
    char   *cnames[NCAT];
    int    nfields, nlines, ncat;

    /* Startup of the main program */
    main_proc()
     {
         register FILE *file;
         char answ[NMEMBERS];
         register c;

         if((file = fopen("questions.dat", "r")) == 0)
         {
            perror("questions.dat");
            panic("can't read the questions database");
         }
         while((c = fgetc(file)) > 0 && c != '\n')
         {
            register char *fp = db[0][nfields++];
            do
                *fp++ = c;
            while((c = fgetc(file)) > 0 && c != '\n' && c != '\t');
            *fp = 0;
         }
        nlines = 1;
        while((c = fgetc(file)) > 0)
        {
            register n;
            for(n = 0; n < nfields; n++)
            {
                register char *sp = db[nlines][n];
                while(c != '\n' && c != '\t' && c > 0)
                    *sp++ = c;
                *sp = 0;
            }
            ++nlines;
        }
        gid = pg_create("twenty_questions");
        if(gid.site == 0)
            panic("can't create the process group!");
        nmembers = NMEMBERS;
        pg_monitor(gid, tmon, 0);
        t_wait(&mcount);
        tw_init();
        begin
        {
            address addrs[2];
            register message *mp;
            register nrep;

            printf("[%d]: %d members, %d fields, %d lines in db, ncat %d\n",
                my_number, NMEMBERS, nfields, nlines, ncat);
            addrs[0] = gid;
            addrs[0].entry = TWENTY_DB_INIT;
            addrs[1] = NULLADDRESS;
            mp = msg_genmsg(TWENTY_DB, db, FTYPE_CHARS, nlines * STRLEN * NFIELDS);
            nrep = CBCAST_EX(addrs, mp, ALL, answ, 1, (adddress*)0);
            msg_delete(mp);

            printf("%d members acknowledged initialization\n", nrep);
        }
  }

/* Startup of a sub-program */
join_proc()
  {
        register message *mp = msg_newmsg();
        gid = pg_lookup("twenty_questions");
        if(gid.site == 0)
            panic("pg_lookup failed");
        if(pg_join(gid, mp) != 0)
            panic("pg_join failed");
        msg_delete(mp);
  }

/* sub-program reception of a database */
twenty_db_init(mp)
```

```
register message *mp;
{
     int dblen;
     char *dbinit;

     dbinit = msg_getfield(mp, TWENTY_DB, 1, &dblen);
     bcopy(dbinit, db, dblen);
     for(nfields = 0; db[0][nfields][0]; nfields++)
        continue;
     for(nlines = 1; db[nlines][0][0]; nlines++)
        continue;
     tw_init();
     printf("[%d]: %d fields, %d lines in db, ncat %d0,
        my_number, nfields, nlines, ncat);
     reply(mp, "+", FTYPE_CHARS, 1);
}

/*
 * Compute various stuff from db and from view:
 *      nfields = fields in db
 *      nlines = length in lines of db
 *      ncat = number of query categories
 *      my_number = internal 'id' of this process: 0..NMEMBERS-1
 * In 'H' query mode, process my_number=m is responsible for lines l s.t. l mod m = 0
 * In 'V' query mode, this process is responsible for row r s.t. r mod m = 0.
 * Program will not function at all with fewer than NMEMBERS instances running.
 */
tw_init()
{
     register n, c;
     register pgroup *pg = pg_getview(gid);

     if(pg == 0)
        panic("pw_init");
     ncat = 1;
     c = 1;
     for(n = 1; n < nlines; n++)
        if(strcmp(db[n][0], db[c][0]))
        {
           cnames[ncat++] = db[n][0];
           c = n;
        }
     for(n = 0; n < pg->pg_nmemb; n++)
        if(cmp_address(&pg->pg_alist[n], &my_address) == 0)
        {
           my_number = n;
           break;
        }
}

/*
 *
 */
```

```
twenty_query(mp)
  register message *mp;
  {
      register cat, class, f, n;
      register char *query, *heading;

      cat = *(int*)msg_getfield(mp, TWENTY_CAT, 1, (int*)0);
      cat %= ncat;
      class = *(int*)msg_getfield(mp, TWENTY_CLASS, 1, (int*)0);
      query = msg_getfield(mp, TWENTY_QUERY, 1, (int*)0);
      heading = query;
      while(*query != '=')
          ++query;
      *query++ = 0;
      for(f = 0; f < nfields; f++)
          if(strcmp(db[0][f], heading) == 0)
              break;
      /* In H mode, everyone answers.  In V mode, only one answers */
      switch(class)
      {
          char *answ;
          int count;

        case 'H':
          answ = 0;
          count = 0;
          if(f == nfields)
              answ = "F";
          else for(n = 1; n < nlines; n++)
              if(strcmp(db[n][0], cnames[cat]))
                  continue;
              else if(count++ % NMEMBERS == my_number)
                  if(strcmp(db[n][f], query) == 0)
                      answ = (answ && *answ != 'Y')? "?": "Y";
                  else
                      answ = (answ && *answ != 'N')? "?": "N";
          reply(mp, answ, FTYPE_CHARS, 1);
          break;

        case 'V':
          if(f % NMEMBERS != my_number)
              break;
          if(f == nfields)
              answ = "F";
          else for(n = 1; n < nlines; n++)
              if(strcmp(db[n][0], cnames[cat]))
                  continue;
              else if(strcmp(db[n][f], query) == 0)
                  answ = (answ && *answ != 'Y')? "?": "Y";
              else
                  answ = (answ && *answ != 'N')? "?": "N";
          reply(mp, answ, FTYPE_CHARS, 1);
          break;
          default:
            reply(mp, "*", FTYPE_CHARS, 1);
            break;
        }
    }
```

Here is the question-answer program that the user sees:

```
/*
 *    Front end program for playing twenty questions
 */

#include "cl.h"
#include "twenty.h"

int    verbose;
int    CLIENT_PORT;

main(argc, argv)
  char **argv;
  {
      int ask_questions();

      while(argc-- > 1)
        switch(**++argv)
        {
          case '-':
            switch(*++*argv)
            {
              case 'v': ++verbose;   continue;
              default:  printf("-%c: unknown option\n", **argv); continue;
            }
          case '0': case '1': case '2': case '3': case '4':
          case '5': case '6': case '7': case '8': case '9':
            CLIENT_PORT = atoi(*argv);
            continue;
        }

      /* Connect to ISIS */
      isis_init(CLIENT_PORT);
      pg_init();

      /* Runs as a task */
      t_fork_delayed(ask_questions, 0);

      forever
      {
        run_tasks();
        isis_read();
      }
  }

ask_questions()
  {
```

```
int cat, class;
char string[120];
register char *sp;
register c;
address addrs[2];

addrs[0] = pg_lookup("twenty_questions");
if(addrs[0].site == 0)
   panic("twenty-questions asker -- can't connect to database program");
addrs[0].entry = TWENTY_QUERY;
addrs[1] = NULLADDRESS;
printf("Welcome to... twenty questions!\n");
print("Enter a random number: ");
sp = string;
while((c = getchar()) != '\n')
   *sp++ = c;
*sp = 0;
cat = atoi(string);
printf("Enter H:query or V:query...\n");
forever
{
   print("Question? ");
   c = getchar();
   if(c <= 0)
      break;
   class = c;
   while((c = getchar()) != '\n')
      if(c == ':')
         break;
   sp = string;
   while((c = getchar()) != '\n')
      if(c != ' ' && c != '\t')
         *sp++ = c;
   *sp = 0;
   if((class != 'H' && class != 'V') | strlen(string) == 0)
      printf("Enter H:cat=value or V:cat=value...\n");
   else
   {
      register message *mp;
      register nwant, nrep;
      char answ[20];

      nwant = (class == 'H')? 1: ALL;
      mp = msg_genmsg(TWENTY_CAT, &cat, FTYPE_LONG, sizeof(int),
         TWENTY_CLASS, &class, FTYPE_LONG, sizeof(int),
         TWENTY_QUES, string, FTYPE_CHARS, strlen(string)+1,
         0);
      nrep = CBCAST(addrs, mp, nwant, answ, 1, (adddress*)0);
      answ[nrep] = 0;
      printf("\t%s\n", answ);
   }
}
printf("Bye.\n");
```

```
        exit(0);
    }
```

3.1. Getting fancy

The above twenty questions program is not really very fancy: it doesn't restart itself very automatically. Here is a much improved version that automatically starts up NMEMBER+NSTANDBY copies of itself and brings up a new standby after each failure. A standby takes over as a member instantly, so the number of members in this example should never drop below NMEMBERS. (If it does, however, the twenty questions program shown below would abort itself and qa would get 0 responses to all its queries -- a better solution to this is proposed below, but it involves changing qa too).

We made a slight change to the qa-twenty-questions interface in this version: it returns a two-byte answer to queries indicating "who" gave the answer (a number 0..NMEMBER-1) and what the answer they give was. The idea is that even though the task assignment may vary, a caller would always get exactly one answer from each virtual member.

```
/*
 *      A fancier twenty questions program
 */

#include "cl.h"
#include "twenty.h"

int     main_proc(), join_proc(), twenty_query(), hello();
int     start(), next_line(), restart_xfer();
address gid, atoaddr(), pg_lookup();

int     must_join;          /* Flag: this process must join */
int     my_number;          /* Virtual member number, see below */
int     CLIENT_PORT;

#define NMEMBER         5       /* Wants this many members */
#define NSTANDBY        2       /* This many hot standbys */

char    db[NLINES][NFIELDS][STRLEN];
char    cnames[NCAT][STRLEN];
int     nfields, nlines, ncat;

main(argc, argv)
  char **argv;
  {
        while(argc-- > 1)
           switch(**++argv)
           {
             default:
             badarg:
               panic("Bad argument: <%s>\n", *argv);

             case '-':
               switch((*argv)[1])
               {
                 case 'j': ++must_join; continue;
                 default:  goto badarg;
```

```
            }
          case '0': case '1': case '2': case '3': case '4':
          case '5': case '6': case '7': case '8': case '9':
            CLIENT_PORT = atoi(*argv);
            continue;
        }

    /* Connect to ISIS, then fork off appropriate procedure */
    isis_init(CLIENT_PORT);
    allow_xfers(start, next_line, restart_xfer);

    isis_entry(TWENTY_QUERY, twenty_query, "twenty_query");
    isis_entry(TWENTY_HELLO, hello, "hello");


    if(must_join == 0)
       t_fork_delayed(main_proc, (char*)0, (message*)0);
    else
       t_fork_delayed(join_proc, (char*)0, (message*)0);

    /* Now enter ISIS main loop */
    forever
    {
       run_tasks();
       isis_read();
    }
 }

static  pgroup cur_pgview;

/*
 * Monitor changes to view... all members see
 * the same view, so the coordinator can be selected
 * as the first (==oldest) listed member.  The coordinator
 * does restarts as needed.  The first view is passed
 * in manually after pg_create() but treated just like
 * any other.  Only starts one process (if any) per
 * invocation, but since each start will change the view,
 * keeps doing this until enough members are running.
 */
tmon(pg)
 pgroup *pg;
 {
      address gid;

      gid = pg->pg_gid;
      cur_pgview = *pg;

      /* Repartition the database based on new view */
      work_partition(pg);

      /* Coordinator is the oldest member of the group */
      if(cmp_address(pg->pg_alist, &my_address) == 0)
```

```
        {
           if(pg->pg_nmemb < NMEMBER+NSTANDBY)
              start_one();
        }
  }

#define TWENTY  "/fs/moose/b/isis/client/twenty"

char    *jargs[]
={
        "twenty", "-j", 0, 0
};

/*
 * Start new program.  If a failure takes place during this call, it
 * either completes first and the member is seen to join before the
 * failure is seen, or the failure is seen first but the join won't
 * occur -- a nifty use of virtual synchrony to avoid a complicated
 * mess of figuring out if a restart was in progress and how it
 * terminated!
 */
start_one()
  {
        static sno;
        static site_id sid[2];
        address pname;
        register site_id *sp;
        register nsites;
        sview *v, *sv_getview();
        char client[30];

        /*
         * Pick a site to start the thing at, try to distribute processes
         * over sites in a reasonably uniform manner so all won't run at
         * the same place.  v->sv_slist[sno] is the site we settled on.
         */
        v = sv_getview();
        for(sp = v->sv_slist; *sp; sp++)
           continue;
        nsites = sp-v->sv_slist;
        if(sno >= nsites)
           sno = 0;
        *sid = v->sv_slist[sno];

        sprintf(client, "%d", CLIENT_PORT+3*sno);
        jargs[2] = client;
        r_exec(sid, TWENTY, jargs,  (char**)0, "isis", "nullpass", &pname);
        if(pname.site == 0)
           panic("Can't rexec 'twenty' at site %d/%d\n",
              SITE_NO(*sid), SITE_INCARN(*sid));
  }

/* In case of an interrupted state transfer, restart where it left off */
```

```
restart_xfer(bno)
 {
      return(bno);
 }

/* Startup of the main program */
main_proc()
 {
      pgroup *pg_getview();
      register FILE *file;
      char answ[NMEMBER];
      register c, n;

      if((file = fopen("questions.dat", "r")) == 0)
      {
         perror("questions.dat");
         panic("can't read the questions database");
      }
      do
      {
         register char *fp = db[0][nfields++];
         while((c = fgetc(file)) > 0 && c != '\n' && c != '\t')
            *fp++ = c;
         *fp = 0;
      }
      while(c != '\n' && c > 0);
      nlines = 1;
      do
      {
         for(n = 0; n < nfields; n++)
         {
            register char *sp = db[nlines][n];
            while((c = fgetc(file)) != '\n' && c != '\t' && c > 0)
               *sp++ = c;
            *sp = 0;
         }
         if(*db[nlines][0])
            ++nlines;
      }
      while(c > 0);
      ncat = 0;
      c = 0;
      for(n = 1; n < nlines; n++)
         if(strcmp(db[n][0], db[c][0]))
         {
            strcpy(cnames[ncat++], db[n][0]);
            c = n;
         }
      /* Now start things by creating the group... */
      gid = pg_create("twenty_questions", 0);
      if(gid.site == 0)
         panic("can't create the process group!");
```

```
        /* Set up to monitor changes */
        pg_monitor(gid, tmon, (char*)0);

        /* First view won't get sent to pg_monitor, so send it manually */
        tmon(pg_getview(gid));
  }

/*
 * This sets up to start a state transfer; all current members participate
 * Since all see the same current pgroup view (copied to the side in the
 * tmon routine), just copy the site list from the view into the alist
 * provided; all do this in parallel and all see the same view, so all
 * use the same alist. This is the simplest way to generate the alist.
 * we could also have copied msg_getdests(mp). There is no obvious reason
 * to favor one over opposed to the other here. (The dests field will have
 * been expanded by now, of course.)
 */
start(mp, who, gid, ap)
  register message *mp;
  register address *ap;
  address who, gid;
  {
        address *pg = cur_pgview.pg_alist;
        do
            *ap = *pg++;
        while(ap++->site);
  }

/*
 * Send one line at a time, which is pretty inefficient (too small), but for
 * purposes of the demo illustrates a multi-block transfer. Actually, should
 * send the whole db at once, since it is really not very large.
 */
next_line(line, buffer, type, len)
  char **buffer;
  int *type, *len;
  {
        if(line >= nlines)
            return(-1);
        *buffer = db[line][0];
        *type = 0;
        *len = nfields*STRLEN;
        return(0);
  }

/* Get a line, sent above */
gotline(line, buffer, len)
  char *buffer;
  {
        bcopy(buffer, db[line], len);
  }

/* QA uses this to find out how big the database is */
```

```
hello(mp)
  register message *mp;
  {
       if(my_number)
           return;
       reply(mp, db, FTYPE_CHARS, NFIELDS*STRLEN);
  }

/* Startup of a sub-program */
join_proc()
  {
       register message *mp = msg_newmsg();
       register rv,c, n;
       int gotline();

       gid = pg_lookup("twenty_questions");
       if(gid.site == 0)
           panic("pg_lookup failed");
       if((rv = pg_join_and_xfer(gid, mp, gotline, X_BIG)) != 0)
           panic("pg_join failed: rv %d", rv);
       msg_delete(mp);
       for(nfields = 0; db[0][nfields][0]; nfields++)
           continue;
       for(nlines = 1; db[nlines][0][0]; nlines++)
           continue;
       ncat = 0;
       c = 0;
       for(n = 1; n < nlines; n++)
           if(strcmp(db[n][0], db[c][0]))
           {
               strcpy(cnames[ncat++], db[n][0]);
               c = n;
           }
       /*
        * same trick as above, although this process is unlikely to be the
        * coordinator yet.
        */
       pg_monitor(gid, tmon, (char*)0);
       tmon(pg_getview(gid));
  }

/*
 * Each time the group view changes, divide up the work.
 * Crash the program if the number of members drops too low
 * (shouldn't happen)
 * The idea is to have each process know a "virtual" number
 * that defines its responsibility for some chunk of the database
 * if my-number is i, this process handles 'V' mode queries for
 * columns that, mod NMEMBERS, have index i, and H mode queries for
 * rows that, mod numbers, have index i.
 */
work_partition(pg)
  register pgroup *pg;
```

```
{
     register address *ap;
     static was_up;

     if(was_up && pg->pg_nmemb < NMEMBER)
        panic("Can't tolerate more than %d simultaneous failures!", NSTANDBY);
     else if(pg->pg_nmemb >= NMEMBER)
        ++was_up;
     for(ap = pg->pg_alist; ap->site; ap++)
        if(cmp_address(ap, &my_address) == 0)
        {
           my_number = ap-pg->pg_alist;
           if(my_number >= NMEMBER)
              /* Standby's get negative numbers */
              my_number = NMEMBER-my_number-1;
           return;
        }
     panic("work_partition -- I'm not in the alist (never happens)");
}

/*
 *
 */
twenty_query(mp)
  register message *mp;
{
     register cat, class, f, n, comp;
     register char *query, *heading;

     query = msg_getfield(mp, TWENTY_QUES, 1, (int*)0);
     if(query == 0 | ncat == 0)
     {
        print("BAD ");
        pmsg(mp);
        snd_reply(mp, "*");
        return;
     }
     cat = *(int*)msg_getfield(mp, TWENTY_CAT, 1, (int*)0);
     cat %= ncat;
     class = *(int*)msg_getfield(mp, TWENTY_CLASS, 1, (int*)0);
     heading = query;
     while(*query != '=' && *query != '>' && *query != '<' && *query)
        ++query;
     comp = *query;
     *query++ = 0;
     for(f = 0; f < nfields; f++)
        if(strcmp(db[0][f], heading) == 0)
           break;
     /* In H mode, everyone answers.  In V mode, only one answers */
     switch(class)
     {
        char *answ;
        int count;
        case 'H':
          if(my_number < 0)
          {
              /* Hot standby's don't send responses */
              reply(mp, (char*)0, 0, 0);
              break;
          }
          answ = 0;
          count = 0;
          if(f == nfields)
              answ = "F";
          else for(n = 1; n < nlines; n++)
          {
              if(strcmp(db[n][0], cnames[cat]))
                  continue;
              else if(count++ % NMEMBER == my_number)
              {
                  if(compare(comp, db[n][f], query) == 0)
                      answ = (answ && *answ != 'Y')? "?": "Y";
                  else
                      answ = (answ && *answ != 'N')? "?": "N";
              }
          }
          if(answ == 0)
              answ = "*";
          snd_reply(mp, *answ);
          break;

        case 'V':
          if(my_number < 0)
              break;
          if(f % NMEMBER != my_number)
              break;
          answ = 0;
          if(f == nfields)
              answ = "F";
          else for(n = 1; n < nlines; n++)
              if(strcmp(db[n][0], cnames[cat]))
                  continue;
              else if(compare(comp, db[n][f], query) == 0)
                  answ = (answ && *answ != 'Y')? "?": "Y";
              else
                  answ = (answ && *answ != 'N')? "?": "N";
          if(answ == 0)
              answ = "*";
          snd_reply(mp, *answ);
          break;

        default:
          snd_reply(mp, '*');
          break;
      }
  }
```

```
/*
 * This version sends two-part replies: the index of the respondent and
 * the answer from that respondent. Caller will get exactly one answe
 * from each respondent as long as the number of processes running is
 * at least NMEMBERS. See discussion below for the case of too many
 * failures to tolerate.
 */
snd_reply(mp, rep)
  register message *mp;
  {
      char answ[2];
      answ[0] = my_number;
      answ[1] = rep;
      reply(mp, answ, FTYPE_CHARS, 2);
  }

/* String comparison, implements numeric scaler comparisons too */
compare(op, s1, s2)
  char *s1, *s2;
  {
      register n1, n2;
      if(op != '<' && op != '>')
         return(strcmp(s1, s2));
      n1 = atoi(s1);
      n2 = atoi(s2);
      if(op == '<')
         return(n1 >= n2);
      return(n1 <= n2);
  }
```

We promised to explain how we could have handled the number of members dropping below NMEMBER a bit more gracefully. Notice that all the members would detect this situation, its just that they panic in this example instead of doing anything. A better solution is for the group to reply "unavailable" One member would have to do "double duty" and cover for the missing member(s) in the single-reply query mode, or the QA program would hang in that case. Meanwhile, the coordinator is frantically bringing up new members, so with luck the situation wouldn't persist for long. A qa program that gets an unavailable response would have to wait a few seconds and retry.

1. Synopsis

A toolkit routine for transferring state from a process group to a process that is joining it.

2. Interface

```
#include <isis/cl.h>

        /* Client side */
        isis_init(0);
        . . .
        join_and_xfer(gid, mp, routine, size)
          address gid;
          message *mp;
          int (*routine)();

        /* Server side */
        allow_xfer(start_routine, data_routine, restart_routine)
          int (*start_routine)();
          int (*data_routine)();
          int (*restart_routine)();
```

3. Discussion

The state transfer tool is normally used by a process that wishes to join an existing process group without preventing clients from using the group, but needs a copy of some state information to begin functioning normally. The tool "hides" the join and state transfer event so that' clients see this as an instantaneous transition.

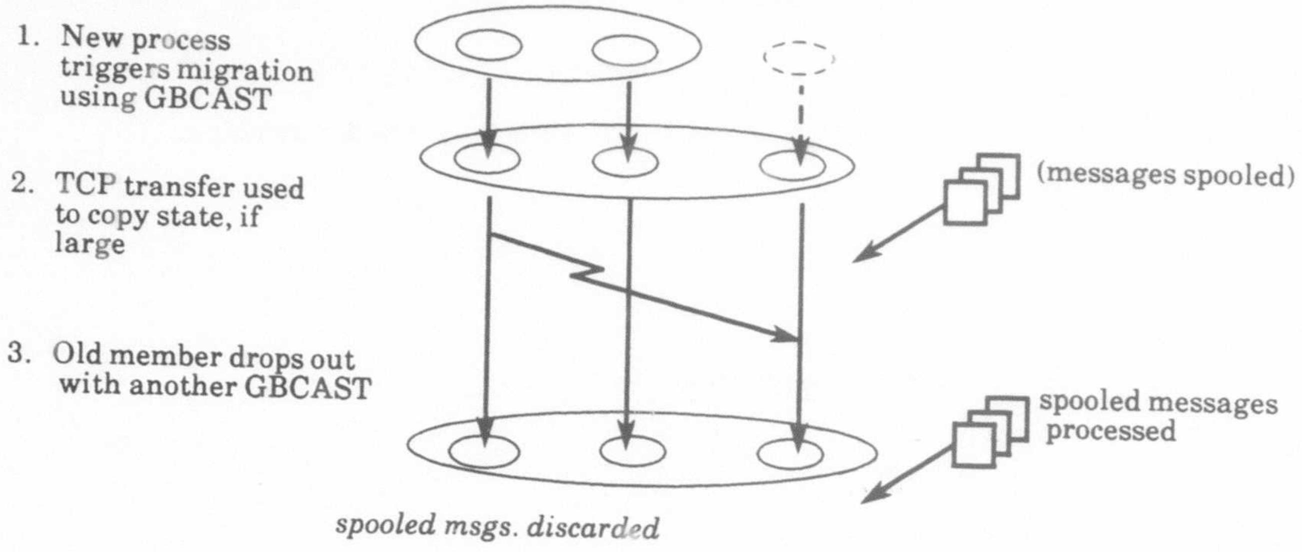
4. Whats a state?

The tool assumes that processes can represent the state of their computation in some number of blocks, which can have arbitrary and variable size. The programmer must somehow write code that lets the tool "read" this state, one "block" at a time. For example, in the twenty questions program, the state is basically the contents of the 'db' data structure -- everything else can be computed or obtained from things like the process group view. In a transactional application, on the other hand, the state should include locking information and output of uncommitted transactions. So, if you use transactions you either have a difficult state packaging problem to overcome (since that tool won't give you this information!) or must do the transfer when nothing that matters is running -- for example, by acquiring read locks on the transactional files before starting the transfer.

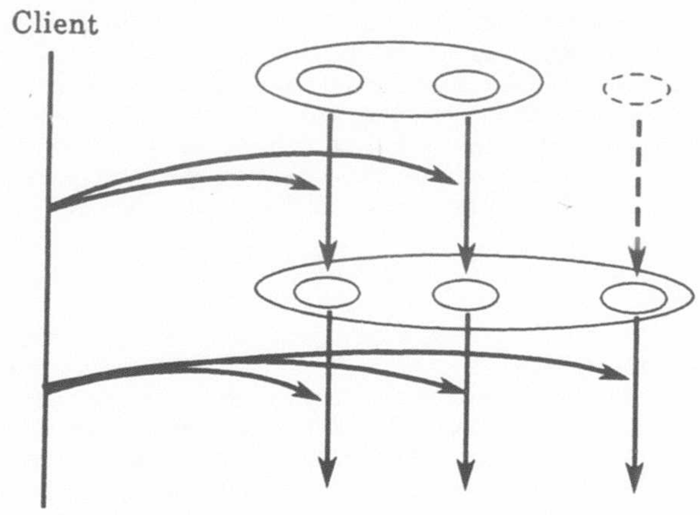
5. How to use the tool

The process wishing to join the group invokes join__and_xfer(), specifying the group to join, a message for the join_verifier (see pg_join() in PGROUPS(TK)) and a state reception routine. The size argument indicates if the transfer will be a big one (X_BIG) or a small one (X_SMALL). A large transfer is done using a TCP stream channel for high performance and would normally require that multiple data blocks be computed and copied. A small state transfer is assumed to fit into a single message, which could be fairly large. In this case, the overhead of a connection set-up is avoided, but on the other hand, the data is transferred by ISIS and hence the throughput is quite a bit lower than using TCP.

Join_and_xfer operates much like pg_join(gid, mp). Assuming that the join succeeds, however, the transfer tool runs a coordinator-cohort algorithm in which the action routine repeatedly requests blocks of state from the generation routine and delivers them to the reception routine. The client's reception routine is invoked as routine(bn,data,blen); where bn is the block number


1. New process triggers migration using GBCAST
2. TCP transfer used to copy state, if large
3. Old member drops out with another GBCAST
(messages spooled)
spooled messages processed
spooled msgs. discarded

State transfer is actually a 3-step algorithm



Clients view state transfers and migration as an instantaneous event

being transferred, data is a pointer to the data, and blen is the length in bytes of this block. If a failure occurs and the transfer restarts, the client will see the block numbers reset to 0 (a future refinement will allow the transfer to resume with the next block in sequence, but this is not yet implemented). When the transfer is completed, join_and_xfer() returns 0 if it succeeded and an error code from pr_errors.h otherwise.

On the side of the group being joined, life is a bit more complex. To allow transfers, the members of this service must all invoke the allow_xfer() procedure, specifying the routines that are to be called when a transfer is started and to obtain each block of state. These routines are invoked as follows.

```
start_routine(mp, who, gid, alist)
  message *mp;
  address who, gid, alist[MAX_PROCS];
```

If the join_verifier has validated the join attempt (see PGROUP(TK)), the start routine is called to compute the set of group components that will participate in this transfer. The routine should fill in this alist, for example by copying pg->pg_alist from the current view of process group *gid*. All members will observe the same pgroup view when the start_routine is called. The argument *mp* is a pointer to the message from the join_and_xfer request and the argument *who* specifies the address of the process that is joining.

```
rval = data_routine(bno, data, type, len)
  int bno, *type, *len;
  char **data;
```

The data routine is responsible for computing block *bno* of a transfer and returns a pointer to a region containing the data and its length through the data and len arguments. The type field should be set using the types known to the MSG_EDIT system, and is used to byte-swap the data being transferred if necessary. The return value should be -1 if the transfer has terminated (there is no block corresponding to offset *bno* and 0 if the block has been computed.

After a failure a transfer can resume either at the first block (bno = 0) or the next block after the last successfully transferred one or any block number in between. The routine

```
start_at = restart_routine(bno)
```

will be invoked with the number of the last transferred block + 1, and should return the next block number to use. For most applications this routine either returns *bno* if it is possible to just continue with the next block, or 0, say if the block sizes or contents could depend on the sender, even though several processes can send the state. (The recipient would have to detect the discontinuity and cleanup from the interrupted transfer if the application requires some sort of cleanup action before restarting at block 0.)

6. Interactions with other tools

This version of the state transfer tool doesn't automatically transfer the state of the toolkit routines, such as the semaphore state. The semaphore tool provides a way to generate a "block" of state information and read it in remotely. The state transfer tool itself will be integrated with the transaction tool in such a manner as to ensure that the transfer occurs only when there are no write locks active in the participating processes. We plan to provide some sort of state transfer option for the transaction file and core-data structure managerment tools too, but these are still undergoing design.

1. Synopsis

A package of routines for obtaining and monitoring the site status data structure.

2. Interface

```
    #include <isis/cl.h>

            isis_init(0);

            . . .

            sview *sv_getview()

            sv_monitor(routine, arg)
              int (*routine)();
              char *arg;

            sv_monitor_cancel(routine, arg)
              int (*routine)();
              char *arg;
```

3. Discussion

This package permits a program to access and monitor the site-view data structure maintained by the ISIS failure detector. The fields of a site-view are: sv_slist[] (the site-id's of the operational sites, null-terminated), sv_incarn[i] which gives the current incarnation number for site i, sv_failed, which lists processes that failed when the view last changed, and sv_recovered, which lists those that recovered. The latter two are both bitvecs. The data structure itself is defined in pr_sviews.h.

The routines are as follows:

a) sv_getview() obtains the most current site view.

b) sv_monitor() causes the specified routine to be invoked as

```
            routine(sv, arg)
              sview *sv;
              char *arg;
```

each time the site-view changes.

c) sv_monitor_cancel() cancels an sv_monitor() request. The arguments must match those for the sv_monitor(). It fails if the monitor request is unknown.

Some things that you can assume about site-views include: the sv_slist[] entries are ordered according to decreasing "age": the first sv_slist[] entry is the site that has been up longest and the last is the site that came up most recently. The bitvecs (see BITVEC(TK)) sv_failed and sv_recovered indicate which site's have undergone a failure/recovery (never both) since the last view was committed. Several sites could change status in a single change of site-view. Finally, the sv_incarn[] vector gives a quick way to check the incarnation number of a particular site in order. The sv_incarn[] entry for a site will contain an "illegal" value if the site is down.

1. Synopsis

An overview of the light weight task facility provided by ISIS.

2. Interface

```
#include <isis/cl.h>

        t_fork_urgent(routine,arg,mp);

        t_fork_delayed(routine,arg,mp);
          int (*routine)();
          char *arg;
          message *mp;

        value = t_wait(cond);
          int value;
          condition *cond;

        t_sig_urgent(cond,value);

        t_sig_delayed(cond,value);
          condition *cond;
          int value;

        run_tasks();
```

3. Discussion

Although normal UNIX systems provide only a single thread of control per process, it has been convenient in ISIS to pretend that each process consists of a set of light-weight tasks that share a single address space. The routines in the task utility provide the ISIS client with access to the light-weight tasking mechanism we implemented for this purpose. The mechanism is a fake in several respects: even though a process can have many tasks, the entire process blocks if a blocking system call is performed. Moreover, there is no true concurrency in the scheme, although it can sometimes look as if there is. In particular, the programmer must be very wary about possible race conditions whenever a task might suspend.

A task is a thread of control having its own stack and registers, but sharing global variables (including static ones) with other tasks. The stack of a task is limited in size, currently to 8k bytes, and if this limit is exceeded, the error will not be detected. However, 8k bytes is really quite a substantial amount of space unless recursion is attempted. The system task is the one that was running when the process started up, and it has an unlimited stack, so you may want to take advantage of this if an algorithm definitely needs more than 8k stack space. Certainly, programmers who work with tasks will need to avoid allocating large amounts of data on the stack or recursive algorithms.

Like coroutines, tasks can employ signals and condition variables to block while waiting for one-another. The basic interface is as follows:

t_fork_urgent(routine,arg,mp)
: The caller is suspended in a runnable state and a new task is created, invoking the designated routine as:

```
        routine(arg)
          char *arg;
```

The new task continues to execute until the routine returns, at which point the task terminates and resources it used (stack, register save area) are freed for use by other tasks. The message pointer should normally be 0. If it is non-zero, then msg_increfcount(mp) will be called immediately and msg_delete(mp) when the task exists. This is used in the system itself when invoking a routine after message reception occurs.

t_fork_delayed(routine,arg,mp)

Same as t_fork_urgent, except that the new task is placed on the runqueue and the caller continues executing.

value=t_wait(cond)

Cond must be a variable of type *condition*. It should be initialized to zero explicitly or allocated in a global or static memory location. The caller suspends in a waiting state until a signal is applied to the condition variable.

t_sig_urgent(cond,value)

The first task waiting on cond is awakened and passed the designated value. The caller is suspended in a runnable state on the runqueue.

t_sig_delayed(cond,value)

The first task waiting on cond is marked as runnable and placed on the runqueue. When it gets to run, it will receive the indicated value.

run_tasks()

This routine must be called from the system task to run tasks for a while. It returns when there are no more runable tasks available. For example, the isis "main_loop" loops, first calling isis_read() and then run_tasks().

t_scheck()

This routine checks for stack overflow and calls the panic() routine if one is detected. It is invoked automatically when switching from task to task, but will not detect overflows that happened previously if the stack pointer has returned to a safe area when the switching takes place.

4. Costs

The costs associated with the task mechanism are minimal. A "context switch" can be done about 100,000 times per second on the SUN3 workstation, although task creation may be slow if a new stack area needs to be allocated (re-use of an old one, on the contrary, is nearly free).

5. System task

The task that runs "run_tasks" is special: it is called the system task, and were it to try and call t_wait it might wake up unexpectedly when the light-weight task facility next wants to schedule some other task. Consequently, the system task must never try to block. It is permitted to execute t_fork_urgent, t_fork_delayed, t_sig_urgent, t_sig_delayed, but not t_scheck or t_wait.

6. Caution

It is crucial that the user of this package keep in mind that although a task must suspend itself explicitly using t_wait or t_fork_urgent or t_sig_urgent to be blocked, this can happen as a result of calling routines someone else has coded. The reason this is a major issue is that once a task suspends, in principle any other runnable task could wake up, and this means that global variables could change values or that procedures could be re-entered, although using a different stack and hence with a different set of registers and local variables. *Do not use this package if you do not understand this discussion.*

7. Bugs

The stack overflow check should be done automatically on every procedure call, say by using a modified version of the mcount procedure that gets linked in when a program is compiled with -pg.

1. Synopsis

Nested transaction in ISIS. This mechanism is based on one from ISIS-1, but doesn't require that you program using "resilient objects". The code hasn't all been ported yet, but it should be usable sometime in August.

2. Interface

```
#include <isis/cl.h>

-------- Transaction control --------

        /* Start a new (sub)transaction */
        t_begin(abort_on_failure)
          int abort_on_failure;

        /* Commit a (sub)transaction */
        t_commit();

        /* Abort a (sub)transaction */
        t_abort();

-------- Accessing files (stable storage) --------

        t_sopen(file_name, how, fmode)
          char *file_name;

        t_ssize(file_name)
          char *file_name;

        t_sread(file_name, offset, buffer, len)
          char *file_name, *buffer;

        t_swrite(file_name, offset, buffer, len)
          char *file_name, *buffer;

        t_sfsync(file_name);
          char *file_name;

        t_sclose(file_name)
          char *file_name;

-------- Accessing in-core storage transactionally --------

        t_copen(item_name, how, fmode)
          char *item_name;

        t_csize(item_name)
          char *item_name;

        t_cread(item_name, offset, buffer, len)
          char *item_name, *buffer;

        t_cwrite(item_name, offset, buffer, len)
        char *item_name, *buffer;

      t_cfsync(item_name);
        char *item_name;

      t_cclose(item_name)
        char *item_name;

-------- Concurrency control --------

      t_rlock(alist, item_name, offset)
        char *file_name;

      t_wlock(alist, item_name, offset);
        char *file_name;

-------- Internal, to monitor for commit and abort events --------

      t_monitor(routine)
        int (*routine)();

      t_outcome(tid)
        trans *tid;
```

3. Discussion

t_begin, t_commit, t_abort. Although ISIS normally does not run in a transactional mode, the whole system is compatible with transactions in a way that makes it easy to obtain them, if desired. To turn on "transactional execution", a routine simply calls t_begin() and later, when it terminates, either t_commit() or t_abort(). If a routine is called by another transaction, the result is a nested transaction. The caller that invokes t_begin() should also indicate whether this (sub)transaction should automatically be aborted in the event that the process that did the invocation should crash, or its site should fail. The only case when abort_on_failure should ever be false (0) is when the transaction is being done in a coordinator-cohort computation and some cohort will take over and run the request forward to completion, doing *exactly* what the failed coordinator did (see the various ISIS papers on roll-forward transactional execution for details). Normally, you will thus request abort_on_failure by setting this flag to true (1). If abort_on_failure is false but the transaction is not restarted in this manner, your application is quite likely to hang.

t_open, ... t_close. These routines access a file transactionally, using t_monitor to detect the termination of each transaction or subtransaction automatically. The t_sxxx versions work with disk files and the t_cxxx versions with in-core data structures. Checkpoints are needed to recover from total failures in the latter case; this is automatic when using stable files. They can be called "as is" (as are?), or can be called from the replication package to implement replicated files (the file name should refer to a different copy of the file in each replica manager, of course). If used in this manner, the default broadcast primitive should be CBCAST.

t_rlock, t_wlock. These routines support transaction read (non-exclusive) and write (exclusive) locking, following the standard 2-phase locking protocol. The alist argument indicates where the lock in question lives. Both give what seems to be "all copies" locking, but the rlock algorithm actually is asynchronous, whereas the wlock algorithm is a slower synchronous one. So, read locks are much cheaper than write locks in the case of replicated data. Lock of either kind will be reissued silently if requested more than once. Note: read locks are never "broken" by failures in ISIS. Note: when using wlock on replicated data, take care that all callers of wlock do their wlock calls in the same order, or deadlock can result. For example, you could use ABCAST to implement a group RPC and then have all members call wlock in parallel on their private data, or your could use CBCAST to implement the RPC and then employ a coordinator-cohort algorithm, this time having only the coordinator call wlock and specifying the group's id in the alist argument.

Semaphores can also be used to control access to files and data, but they ignore the transactional scope rules and hence could get you into trouble.

4. Descriptions of internal routines

Transaction ids. When running as a transaction, ctp->task_tid is a pointer to a descriptive structure characterizing the state of the current transaction. This can printed by calling t_print(ctp->task_tid).

t_monitor. This routine is used internally by ISIS to watch for the commit or abort of a transaction that has taken actions in the invoking process. The routine is invoked as:

```
routine(how)
{
}
}
```

where the argument *how* will be one of T_COMMIT1, T_COMMIT2, and T_ABORT. The commit phases are the usual ones for a two-phase commit. If you plan to implement your own transactional storage, then during phase 1, records written by the transaction should be forced to stable storage. If stable storage is not an issue, take no action during phase1 commit. During phase 2, commit records can be written. In a T_ABORT, the effects of the transaction must be rolled back. This is all automatic in the case of the stable storage routines provided by the system.

t_outcome. This routine is used when a site recovers and the stable storage utility discovers that it crashed during the second phase of the commit protocol for some transaction. If all the sites that know what this outcome was are down, it could take quite a while for this routine to complete, and while it is running access to the file the transaction updated is not allowed (both t_rlock and t_wlock will block). On the other hand, the updated file can be accessed using t_read and t_write without acquiring locks if an emergency need to see the contents arises. One would obtain the contents of the file as if the transaction did commit in this case. Since the odds are overwhelming that this is exactly what happened, the behavior that results is probably fine.

5. Recovery

There are several cases:

1. In the case where you arranged for the stable storage routines to be called from the replicated data manager, a facility is provided that will automatically being your copy of the file back into "sync" with any others after recovery. Use the rmgr to determine which of the following cases applies. If your program is the first to recover from a crash of all members of the group, it can use its local copy of the file -- they will already have been restored to a consistent state by the transaction facility. If your program is supposed to rejoin, we currently recommend that the entire file be copied from some process with a copy. A better mechanism will be added someday, but meanwhile this will have to do.
2. When using the in-core storage routines, recovery depends on how the failure occurred. If the failure causes all processes to crash, you must have a checkpoint around in order to recover. Assuming that you do, the recovery sequence is as above, but using the checkpoint. You can make a checkpoint anytime the data is idle, but not during a transaction that has updated it.

3. In the case where in-core transactional storage is being used and a partial failure takes place, the state transfer tool should be used to copy the data from some operational process possessing a replica.

6. Examples

Transactions are easiest to use if you follow very stylized coding conventions. Some examples to illustrate the most common cases follow.

6.1. Non-replicated data

6.2. Replicated data, resilient object style

6.3. Replicated data, CIRCUS style

6.4. Replicated data, using quorums

7. State transfers

This will look something like the semaphore state_xfer_out/in mechanism.

1. Synopsis

What's this virtual synchrony business all about anyhow, and what do I need to do about it?

2. Discussion

All of ISIS is built using a collection of broadcast communication primitives that, if used correctly, provide "virtually synchronous" distributed executions. This idea is one we use throughout ISIS, and it can greatly simplify the design of even very high level software.

Figure 1 shows a conventional distributed execution. Such an execution is characterized by message passing and the discovery of occassional "unexpected" events, such as crashes, timeouts due to system overloads and transient communication failures, reception of new "request" messages while pending requests are still underway, and differences in the perceived system state, from process to process, even when a single event is being observed from multiple perspectives. An environment like this one is difficult to work with -- we call it a completely asynchronous one -- and it provides very little support for the programmer who must implement a distributed application program.

In Figure 2, a virtually synchronous execution is shown. Such an execution has the property that it *looks* to an observer (to a program using ISIS, in particular) as if one event takes place at a time in the system. That is, if a process fails, it looks as if no communication events were active at the time, and everyone monitoring for the failure sees it occur "simultaneously". When a communication action occurs (see BCAST(TK)), failures never seem to take place until the messages have been delivered, and it looks as if no other communication was taking place at the same time. In fact, each message indicates the processes to which it was delivered, even though the message may have been addressed using process groups as destinations and process groups have dynamically varying membership. This has several real advantages from the perspective of the programmer

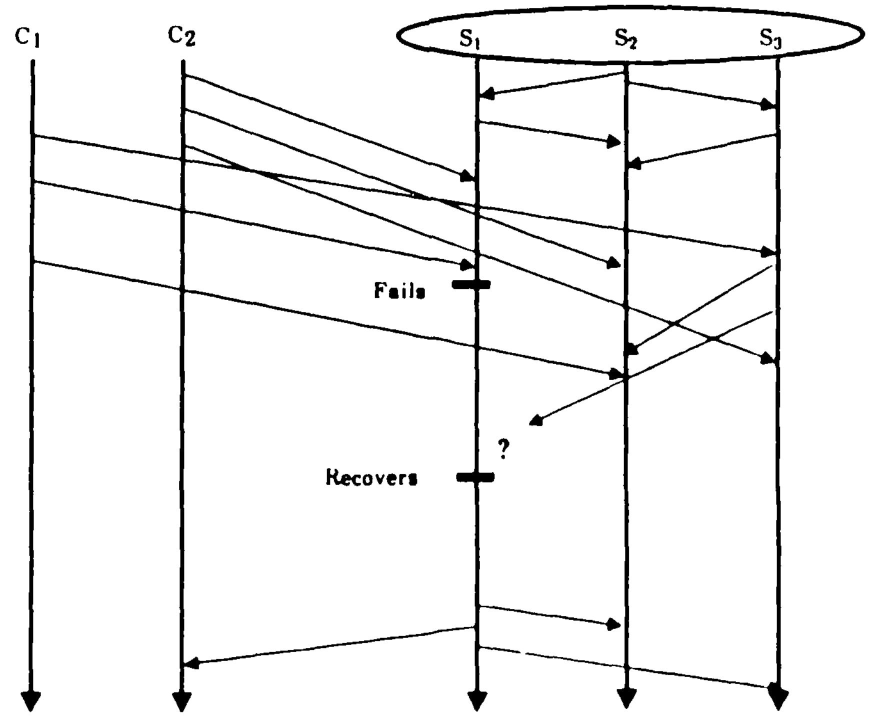


Figure 1: Conventional message-passing picture of a system

who works with ISIS.

One relates to algorithm design. In ISIS, it is possible to deduce the actions that other processes will take by just looking at a message, its destinations, and the "state" of the system at the time the message arrives. For example, this might include the membership of a process group (pg_alist[] in a process group view), the sites that are operational (sl_slist[] in a site-view), the contents of the message, or even user-supplied data structures maintained using the configuration tool (see CONFIG(TK)). In fact, the addresses in a process-group or site view are even ordered according to increasing age, and you can use this in your code. Moreover, all members of a process group receive a broadcast message if any does so -- there is no need to manually make sure that everyone has their copy. Finally, actions initiated by a dead process are terminated before the death is announced... for example, if a process might have been issuing a broadcast or adding a member to a process group or taking some other action when it died, either the action completes before the process failure is observed, or the action never happens at all -- the failure preceded it. Jointly, these aspects really simplify life: they eliminate the chit-chat normally needed when a group of processes receive a message, and let everyone act in a coordinated fashion without taking any actions to achieve the coordination. Of course, it may be necessary for group members to monitor one another, but prepackaged tools like the coordinator-cohort mechanism (COORD(TK)) and the monitoring routines (pg_monitor in PGROUPS(TK) and the watch() routine in WATCH(TK)) make this as easy as possible. Or, one can arrange for the client of a group to take part in getting an action done by simply having all members respond to "their part" of a request, and having the client collate responses, decide if the action really got done, and re-issue the request if necessary.

One implication of virtual synchrony is that most ISIS mechanisms are orthogonal to each other. Hence, you can combine semaphores, state transfers, and replicated data without one mechanism impacting much on the others. Of course, this isn't magical: state transfers while the semaphore is

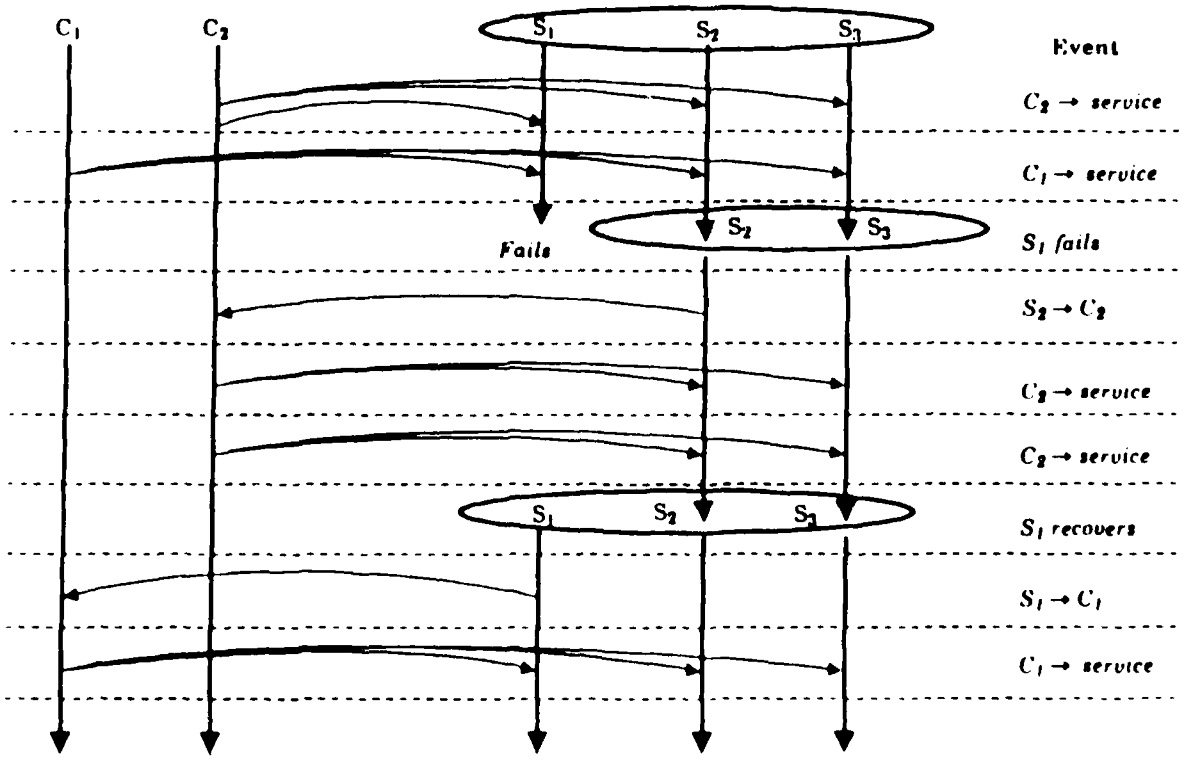


Figure 2: Virtual synchrony picture with a process group and a few clients

in use or while transactions are running can be a bit awkward. But, mechanisms to avoid these problems are provided in most cases, and being added in others.

In addition, several of the data structures that ISIS maintains have properties based on virtual synchrony. As noted above, everyone sees the same sequence of pgroup views and site views (see pg_monitor() and sv_monitor()) and within these views, the sequence of addresses in a pg_alist or site-id's in an sv_slist are ordered according to *decreasing age*. Moreover, the current pgroup view at the time a message is received is the same for all recipients. Thus, one can receive a message, check the current view, and then make a decision in such a way that only one process is responsible for executing the message and all others are backups -- this is how the coordinator-cohort algorithm is built in ISIS. Once you get used to taking advantage of this approach, you will see how it simplifies your code: rather than design a protocol to discuss who should handle a request, you design a simple *local* decision that everyone who receives the request can execute in parallel, in such a way that all reach the same decision without message exchange.

Another advantage to the ISIS approach is that there are genuinely fewer things to worry about when designing code. Basically, you can use a finite state approach. In a given state, your program may:

1. Be waiting for a response to some request (or several, if concurrent tasks are running).
2. Detect a failure.
3. Receive a new request.

But, since all copies of the program see these events in the "same order", and everyone sees every event that concerns them, there is no uncertainty in your code: for each possible event, you simply specify the appropriate actions and you are finished. Unless you omit the case of, say, a failure occurring while your program is waiting for a response from some process, and this causes a crash to occur, code that covers the above cases will cover everything necessary for correct performance in ISIS. In contrast, imagine the uncertainty of executing in a conventional environment! Failures may be detected incorrectly, messages may fail to reach some destination, or arrive out of order, and events may be perceived in different orders by different processes in the system. The simplest actions are fraught with danger. Many programs that can easily be written using ISIS are, for these reasons, nearly impossible to write in any other way!

What does virtual synchrony cost? Well, the cost could be high if you use the most costly broadcast primitive (GBCAST) too casually, and this is a major reason for using the toolkit as much as possible. The tools use the cheapest broadcast they can, and performance will be good when you stick to them -- at least for things that ISIS is good at. These are things like maintaining replicated or recoverable data structures, synchronizing actions, and sending requests to services. On the other hand, bulk data transfers are best accomplished using the state transfer mechanism or other non-ISIS mechanisms. With careful attention to design, performance of an ISIS-based application can be as good or better than for a non-ISIS application.

What is the minimum you need to understand? Basically, the ISIS user has two kinds of decisions to make. One is to decide how to structure the application into process groups and processes and what data structures will be needed. Often, the ISIS toolkit routines can be used to implement this structure, following our tutorial examples, but in many cases you will need to perform "group RPC" requests. This leads to the second decision: when sending a message, you will need to decide whether an answer is needed, and how many answers are needed (0, 1, n, or ALL). You will also need to determine whether the group is actually sensitive to the order in which it receives this type of requests – if so, you should use ABCAST to send the requests, if not CBCAST is preferred. For example, a service that maintains a replicated queue of some sort would probably be accessed using ABCAST: queue order will be the same everywhere if requests arrive everyone in a fixed order, and thats exactly what ABCAST provides. On the other hand, a service that maintains a database and answers questions out of it could normally be accessed using CBCAST performance will be better, and in this case the order in which queries arrive doesn't change the answer that should be given. GBCAST is normally used only in the toolkit routines.

Who should answer a query? The easiest solution might be for everyone to reply (some replies might be of the form "I don't know", indicated by calling reply(mp, 0, 0, 0)). Also, keep in mind that one can reply with fewer than "alen" bytes of reply information (BCAST(TK)). For example, if the reply is a byte string, the the first byte could be a code indicating whether the rest of the reply contains valid data. If you prefer to receive a single reply the coordinator-cohort tool should be used. This has some overhead and the amount of work done by the coordinator should be non-trivial to justify paying this added cost. One situation in which the tool is not recommended arises when the reply will be very large. It might seem like you should use the tool to avoid wasting "space" on replies from processes other than the coordinator. In fact, however, this would be just the situation in which the overhead of the coordinator-cohort algorithm turns out to be largest! The overhead is almost zero, on the other hand, if the coordinator sends the data using a CBCAST to the caller and then replies with a status code, say an integer.

To summarize: virtual synchrony makes the toolkit possible and makes algorithmic design surprisingly easy in ISIS. The benefits are substantial, but the programmer may be expected to make some decisions that could affect performance, and to do this intelligently requires some understanding of broadcast orderings. We strongly recommend that you read the ISIS papers if this applies to you: documentation has a role, as do tutorials, but the papers are much more systematic in attacking this sometimes subtle material.

1. Synopsis

A package of routines implementing a watch facility.

2. Interface

```
#include <isis/cl.h>

        isis_init(0);

        . . .

        /* Watch a process */
        wid = watch(addr, gid, routine, arg)
          address addr, gid;
          int (*routine)();
          char *arg;

        /* Wait for a process to join a process group of which caller is a member */
        wid = watch_for(addr, gid, routine, arg)
          address addr, gid;
          int (*routine)();
          char *arg;

        /* Watch a site */
        wid = site_watch(sid, routine, arg)
          site_id sid;
          int (*routine)();
          char *arg;

        /* Cancel a watch request of either sort */
        watch_cancel(wid)
          int wid;

        watch_dump()
```

3. Discussion

The watch facility is a tool for triggering actions in the event that the status of a process or site should change. In the case of a process, watch() is used to watch for failure, whereas watch_for() is used to watch for recovery. The argument "addr" should give the address of the process to watch (for) and "gid" should be a group to which both the watched process and the caller belongs. In the case of watch(), the gid can be specified as NULLADDRESS, in which case the routine uses a more costly algorithm but can monitor any process at any site in the cluster. If possible, specify a group id; watch is much cheaper in this case. In the case of a site-watch, the behavior of the watch facility depends on the site incarnation number given in the sid. If this number is non-zero, the facility will notify the caller if the designated site fails. If the incarnation is given as 0, however, the facility will notify the caller if the site makes any sort of a transition: from up to down or from down to up. In all cases, a non-zero unique identifier is returned and can be used to cancel the watch later. *Watch returns 0 if the watch is impossible because the event has already taken place. Watch returns -1 if the caller is not a member of the designated group.*

The callback routines are invoked as follows. In the process watch case:

```
        routine(addr, arg);
```

Note that when doing a watch_for(), a parallel watch() might need to be done to detect and handle the case where the watched-for process fails instead of joining the group. Depending on which event occurs, the other event would be canceled when the routine is called.

In the site watch case the call sequence is as follows:

```
routine(sid, arg);
```

Callback can only occur when a message received by the watch subsystem triggers a site-view or process-group view change and the event was being watched for, a callback can take place immediately. This implies that the isis_read() routine was active, hence the caller can assume that callback will not happen "asynchronously" during other computation.

4. Bugs

Watch with NULLADDRESS specified as the gid has not yet been implemented. It will be implemented when the transactional facility is added to ISIS later this summer.