Ensemble Tutorial

Mark Hayden, Ohad Rodeh

July 23, 2002

Abstract

Ensemble is a reimplementation of the Horus reliable group communication system in the Objective Caml programming language. This document describes:

- How to configure and execute the applications included with Ensemble.
- The text-based Ensemble application interface.
- The ML Ensemble application interface.
- The native C Ensemble application interface (*CE*).
- The HOT C interface, a legacy from Horus.

Contents

1 Quick Installation

Several demonstration applications are included with Ensemble. These can give a sense of the kinds of facilities provided by group communication to those who have not used a group communication toolkit before. The demos can also serve as starting points for building new applications. These applications are briefly described here along with how to execute them and the various command-line options and environment variables they use.

1.1 Compiling

Please see the file **ensemble/INSTALL** for instructions on installing Ensemble if you have not done so already.

1.2 Environment Variables

Detailed information is given in Section 3.1 for initializing environment variables. Here we give the bare minimum you need in order to get started. We assume that you will (at least initially) be using the gossip server for processes to locate each other. **ENS_GOSSIP_PORT** must be set to a port number that is not used by other applications. Normally, user applications cannot use port numbers below 1000. **ENS_GOSSIP_HOSTS** must be set to a list of colon-separated host names where the gossip server may be found. If you wish to use port 7500 for the gossip server on hosts "ely" and "natasha," you would set these environment variables as follows (in Unix csh):

```
% setenv ENS_GOSSIP_PORT 7500
% setenv ENS_GOSSIP_HOSTS ely:natasha
```

Throughout this tutorial, we assume you are using the Unix csh or tcsh shell. To set an environment variable in the bash shell, you would do the following:

```
% export ENS_GOSSIP_PORT=7500
% export ENS_GOSSIP_HOSTS=ely:natasha
```

Remember that these must be set for all Ensemble programs you run. You may wish to add the configuration to your **.cshrc** or equivalent.

1.3 Executing Applications

Applications by default require a gossip server process to be running in order to contact each other. Before executing an application, a gossip server must be started one of the hosts listed in **ENS_GOSSIP_HOSTS**:

```
% gossip
  ...
```

On the same or other hosts execute several instances of an application, such as **demo/mtalk**:

```
% mtalk
  ...
```

2 The Programs

Notes:

- please note that warning and error messages printed by Ensemble are not prefixed with the name of the program generating the message, but rather the name of the module.

2.1 Mtalk: Multi-person Talk

This is a multi-person talk demo. As **mtalk** processes are created, they merge into a single group. Input typed at one process is broadcast to the rest of the processes in the group.

2.2 Wbml: Distributed Whiteboard

This is a graphical white board demo. It uses the CamlTk library to implement a graphical user interface[1]. When members are in the same group, lines drawn on one instance are broadcast to the rest of the group, who also draw the lines. It supports the switching of protocols. Initially, wbml has an auto-merge protocol in its stack so the members merge together. This can be removed to disable partition healing. Adding the XFER protocol, causes members to transfer their state on view changes; the TOTAL protocol enable totally ordered communication. Initially, these extra protocols are not included in the protocol stack.

2.3 Ensemble: Text-based Interface

This program provides a text-based interface to the Ensemble group communication facilities. You can run it to directly see what happens in an Ensemble application. You start up the program and it prints out messages decribing changes to the membership of the group. You can type in commands such as “cast hello” which causes Ensemble to broadcast “hello” to the other members of the group, who get “cast hello” printed out. This program can be used as a subprocess of an application for doing basic group communication. The normal usage is to set up pipes to and from the standard input and output of the **ensemble** process.

In order to distinguish different applications that are using this interface to communicate, you may wish to use the **-group_name** option to set the name of the group.

The input of the program must be formatted in text lines as follows:

- [cast **msg**] broadcasts the following message to the group, where **msg** is the remainder of the input line. (Normally, the broadcaster will not receive the message).
- [send **dest msg**] sends a point-to-point message to the rest of the group. **dest** is the endpoint identifier of the group member you wish to send the message to. **msg** is the remainder of the input line.
- [leave] causes the member to leave the group. This will eventually result in an exit message being output and then the **ensemble** process will exit.
- [prompt] prompt for a view change
- [suspect **suspect**] suspect a member, this will cause that member to be expelled from the view.

---

[1] not supported under Windows/NT

- [xferdone] state transfer is complete for this member. If you are using the state-transfer layer, then this actions has effect. Once all members notify the system that state-transfer is complete, a fresh view is created, with the xfer bit set to false. For more information, look up the XFER layer.
- [rekey] rekey the group.
- [protocol **proto_id**] switch the protocol

The output of the program consists of lines in one of the following formats:

- [endpt **endpoint_id**] is output as the first line and only appears once. It gives the name of this application as it will appear in views.
- [view **nmembers my_rank view**] describes a new view of the group. Initially, every member begins in its own singleton view. Other members are added through automatic merging with other views. Members are removed through failure detections. **nmembers** is the number of members of the group. **my_rank** is this member's rank in the new view. **protocol** is the name of the protocol being used. **view** is a space-separated list of the endpoint identifiers of the members of the group.
- [cast **origin msg**] displays a broadcast received from the member of rank **origin**.
- [send **origin msg**] displays a point-to-point message received from member of rank **origin**.
- [exit] notifies that this member has left the group as a result of a previous **leave**. This is the last line output by **ensemble**.

2.4 Gossip: Group Locator Service

This is not really an application. The gossip server works in conjunction with the Ensemble **UDP** communication transport to simulate low-bandwidth gossip broadcast for systems that do not have IP multicast. See the discussion on transports below. The group communication protocols require some "gossipping" mechanism in order to detect and heal partitions in the system. When an application wishes to gossip with other partitions, it broadcasts a message via the **gossip** servers. This sends messages to the **gossip** servers. The **gossip** servers then forward the message to all processes they have heard from recently to simulate a broadcast. When an application is using the **UDP** transport and not the **DEERING** transport (**UDP** is the default), it is necessary for a **gossip** process to be running somewhere in the system.

2.5 Groupd: Membership Service (formerly called Domain)

Normally, Ensemble application groups implement their own group membership protocol. However, they have the option of using the Ensemble membership service implemented by the **groupd** application. **groupd** is a service for managing multiple process groups. It uses a *core* group of Ensemble processes to participate in managing these groups. Clients connect to the service via TCP connections, through which they request to join and leave groups. The service supports a simple protocol through which the clients can obtain virtual synchronous properties. The service also supports weaker properties that give faster membership notifications.

[We emphasize that Ensemble applications can operate independently of a membership service.]

Some of the benefits of using this service are:

- When there are no membership changes, the clients communicate directly between themselves, so the membership service has no affect on performance.
- The service implements group membership for multiple groups. The costs of the group membership protocols (such as failure detection) are shared over the groups.
- Because applications are sharing the same membership service, they see consistent views and failure detections.
- The client part of the protocol for implementing virtual synchrony is simple. Most of the complexity is in the server. This allows client programs to be implemented in languages other than ML, but save much of the programming burden because the servers handle the "hard" group membership protocols. The client TCP interface is described in the Ensemble reference manual.
- Applications that do not need the full virtual synchrony properties can use weaker synchronization protocols and get faster view changes.
- The service allows groups to scale to larger sizes. The membership servers do not need to run on all the hosts on which the clients run, so clients can be on more hosts than are normally supported by Ensemble.

Executing Groupd: In order to run Groupd, set the **ENS_GROUPD_PORT** environment variable to select the TCP port for the service to use. The membership service is executed through the **groupd** application program:

```
% groupd
```

It takes command-line arguments similar to the other Ensemble demonstration programs. Normally, each host runs a server.

Other demo applications use the service when the **-groupd** command-line argument is selected. For example:

```
% mtalk -groupd
```

Note that you must have a **groupd** server running on the same host as mtalk for this to work.

2.6 Perf: Performance Tests

This program includes a variety of performance tests for Ensemble.

Ring: This test is run with the **-prog ring** option. Say that there are n members. Each process first waits until there are n members. It then sends k messages, and waits for $(n-1)k$ messages from other members. It measures the time for this, and does so a number of times to determine the average and variance. This can be done for varying n, k, message size, and protocol.

The time between the rounds is a measure of latency. The total amount of data sent between the rounds is a measure of bandwidth. The total number of messages sent between rounds is a measure of throughput. For good measurements, set the parameters as follows:

measure	k	message size
latency	1	0
throughput	large	0
bandwidth	1	large

Additional command-line arguments (with default values in parentheses):

-n # : number of members (2 members)

-s # : size in bytes of application messages (0 bytes)

-r # : number of rounds (300 rounds)

-k # : messages per round (1 message per round)

These values must be set by all members. All members must use the same values for all of the arguments except message size.

[**TODO: The other performance tests are undocumented.**]

2.7 Rand: Virtual Synchrony Debugging Tool

This demo is used to test Ensemble. It uses simulated communication and introduces random process failures to check for proper behavior of the group membership protocols.

2.8 Fifo: Fifo Communication Debugging Tool

This demo is used to test Ensemble. It uses simulated communication structured in such a way as to trigger bugs in FIFO, reliable communication protocols.

2.9 Armadillo: testing Ensemble security extensions

This program tests Ensemble security features. It has several command line options:

-n # number of endpoints to create

-t # after what threshold to start the test

-prog which security to use? [policy,rekey,exchange,reg,prompt]

-pa simulate partitions?

-net run everything in a single process or run throughout the network

-real_pgp use PGP for authentication? otherwise, simulate it.

-group set the group name

The “exchange” test checks that the Exchange layer functions correctly. For example, running:

```
% armadillo -n 20 -prog exchange
```

will create 20 endpoints with random intial keys. the endpoints should merge into one group after a short while.

The "rekey" test creates a group and once its size is above the threshold it start rekeying it. The test: `Use: armadillo -n 7 -t 7 -prog rekey` will create a group of 7 members and once the group reaches this size, will start to rekey it.

To see what happens when the group partitions use: `armadillo -n 5 -t 3 -prog rekey -pa`. This will create a group of 5 members and start partitioning and remerging the group. Everytime the membership in a group component exceeds 3, the component leader will try rekeying it.

The "policy" test checks that Ensemble respects application trust policies. For example running:

```
% armadillo -n 7 -prog policy
```

will create a static group of 7 processes, numbered 0 through 6, and dynamically change the endpoints trust policies. Ensemble forms subgroups according to the trust relationship. The policies are designed to change in stages:

1. All endpoints trust each other.
2. All endpoints of the same (mod 2) trust each other. That is we have to trust domains: $\{0, 2, 4, 6\}$ and $\{1, 3, 5\}$.
3. All endpoints of the same (mod 3) trust each other. That is we have three trust domains: $\{0, 3, 6\}$, $\{1, 4\}$ $\{2, 5\}$.

The "prompt", and "reg" tests are auxillary tests not related to security.

2.10 Life: Game of Life Demo

This is a graphical version of the *Game of Life* that was originally "invented" by J.H. Conway in 1969 (and was first reported in *Scientific American*, October 1970). The Game of Life is not actually a game: "it is the study of phenomena which can be observed in evolving configurations of populations. One can think of a population as a generation of living and non-living beings. A generation can be modeled by a rectangular grid of cells in which each being occupies exactly one cell and each cell can be either on or off. If a being is alive, then the corresponding cell is on; if the being is dead, then the cell is off. From this point on we refer to beings of a population and cells of the rectangular grid interchangeably." In this implementation, each cell of the grid is implemented by a separate endpoint and all communication is through asynchronous Ensemble communication. Anyway, this program requires the CamlTk library and should be self-explanatory to run (note that you only need to run one Life process: it creates multiple endpoints within the process).

This application was written by Samuel Weber.

2.11 Socktest

This is a simple application that tests the soundness of the lower level Ensemble interface to sockets and IP-multicast.

The current menu is as follows:

- [Join **ipm_addr**] Join a multicast group.
- [Leave **ipm_addr**] Leave a multicast group.

- [Cast **ipm_addr msg**] Send a message to a multicast group.
- [Ttl **num**] set the time-to-live.
- [Loopback **onoff**] set the loopack. If this is on, then messages sent to a multicast group will also be locally received.
- [Sendbuf **size**] Set the send-buffer size in the kernel. Normally, to little space is reserved in the kernel for storing IP packets, therefore, it is common practice to increase it.
- [Recvbuf **size**] Same, as Sendbuf, only for received packets.
- [Nonblock **onoff**] set a socket to blocking or non-blocking mode.

3 Configuration

3.1 Command-line Arguments and Environment Variables

Ensemble applications typically support a variety of configuration parameters. Most of these can be configured through command-line options as well as through setting environment variables. In all cases, command-line options override environment variables. Look in **appl/harg.ml** for the authoritative list of the configuration parameters. Some are listed below as command-line options. The corresponding environment variable for **-group_name** (for example) is **ENS_GROUP_NAME** (the name is capitalized and the '-' is replace with **ENS_**).

-modes arg : Set the default modes for an application to use. The modes are specified giving their names in all-uppercase, each separated by single colons (':') and no white-space.

-udp_port port : set the default UDP port for Ensemble applications. For point to point UDP communication, this is the port number Ensemble first tries to bind to for UDP communication (if it is already in use Ensemble will then fail). It can be set to any value. The default is to let the operating system choose a port to use.

-deering_port port : This is the port that Ensemble will use for Deering IP multicast communication (if enabled). All processes must use the same port number.

-gossip_port port : sets the port that the **gossip** servers use.

-gossip_hosts arg : sets the hosts where applications using UDP communication can look for **gossip** servers. The value should be a colon-separated list of hostnames. The **gossip** server application will only execute on these hosts. Note that you only have to execute a gossip server on one of these hosts: applications will try each of the hosts in turn while looking for a gossip server. However, multiple servers can be executed for increased availability.

-id name : used to give applications unique identifiers. Usually this is set to be your user id. Setting this variable prevents Ensemble applications run by other users from interacting with yours. In case you do want them to interact, you should set their variables to have the same value. If using DEERING IP multicast, their **-deering_port** variable should also be set to the same value.

-groupd_port port : sets the port that the membership **groupd** servers use.

-groupd_hosts arg : sets the hosts that the membership **groupd** servers use. Format is the same as for **-gossip_hosts**.

-groupd : Use the membership service on the local host (see section 2.5. This option may override others.

-group_name name : Set the name of the application's group. **[Currently, only the ensemble application supports this.]**

-key key : Set the key to use for a particular application. All messages sent and received by the application will be authenticated with this key.

-secure : Enable security enforcement. This prevents any insecure communication transports from being initialized.

-add_prop property : Adds a specific property to the Ensemble protocol stack. See Section 5.8 for more information on supported Ensemble properties.

-remove_prop property : The dual of add_prop.

-sock_buf size : The size of socket buffers to request from the operating system. The default size is 52428 (the traditional limit on Unix). If you are using Ensemble in high-performance setting and are experiencing message loss, this is a parameter that should be increased.

-refcount : Enables reference counting of message buffers. The default is to rely on the garbage collector to detect when a message is no longer needed. Setting this will improve performance, but may expose reference counting bugs in Ensemble.

-multiread : Enable multiple reads on sockets. The default is to receive and process one message from the operating system at a time. Setting this will cause all available messages to be read from sockets before processing any of them, which may reduce message loss due to buffer overflow in the operating system.

-pollcount count : The number of times to query the operating system before blocking. Ensemble blocks after checking (via the **select()** system call) the operating system for messages and not finding any. Setting this to 1 will cause Ensemble to block immediately when there are no more messages. Setting this to a large number will cause Ensemble to busy-poll for a longer time before blocking.

-ranking trans: this is used to set the priority of transport mechanisms. This is useful if you wish to use TCP as a transport instead of UDP (the default). Using `-ranking TCP` will cause TCP to be ranked higher than any other transport.

The following configuration parameter can only be set as an environment variable.

ENS_TRACE : enables module initialization tracing. With this set (to any value), modules print out their names as they initialize. This is useful if an exception occurs during initialization because because it enables you to narrow the problem down to one module.

3.2 Transports

[If you are only using regular UDP sockets for communication, then you do not need to read this section.]

Perhaps the most confusing part of running Ensemble applications comes from selecting communication transports. Communication transports are the bottom-most part of Ensemble and are used for sending and receiving messages on a network. There are several ways this can be confusing and often Ensemble cannot detect that there is a problem, so you do not get a warning. For instance, if you configure an application so that one process is using UDP sockets for communication and another is using NETSIM, then the two processes will stall waiting for other processes to communicate with them on their selected medium.

A confusing aspect of transport is that an application typically uses two different kinds of transports: a primary transport and a gossip transport. Normal application communication is all done over the primary transport, which must support point-to-point communication and may also support multicast communication. Communication between different partitions of a group of applications uses the gossip transport which must support "anonymous" multicast communication.

Applications occasionally send “gossip” messages with their gossip transport to the rest of the “world” in order to inform other partitions about their presence. When two partitions learn of each other, they can then merge the partitions together. After they have merged together, they communicate over their primary transport. This gossip-and-merge mechanism is used when applications first start up: an application creates its own singleton group and then merges with any other already existing partitions through gossiping and merging. Thus, if there is a problem with the gossip transport, you will tend to have a bunch of applications in singleton groups that never merge. If there is a problem with the primary transport, the merging will occur, but then the various members will be unable to communicate. This will cause them to repeatedly break into partitions (when they decide that the other members must have failed) and then re-merge again.

The various primary and gossip transports are presented in the following table. The “P” and “G” columns specify whether a transport can be used for primary communication and/or gossip communication.

transport	P	G	description
UDP	√	√	UDP (+ gossip server)
DEERING	√	√	UDP/IP multicast
TCP	√		TCP/IP
NETSIM	√	√	network simulator

The **NETSIM** transports are used only in applications that are simulating the behavior of a group inside a single process. The **rand** and **fifo** demos use this, for instance. All other currently supported modes run over IP. UDP requires running the gossip server. With TCP as a transport, a fully connected mesh of TCP connections is used to move messages between group members. This is not very efficient for multicast messages, however, it can sometimes be useful.

There are several ways to change the communication transports that Ensemble uses. These are listed below in order of highest “precedence.”

1. Command-line argument: with the **-modes** argument (see the command-line argument documentation).
2. Application setting: a particular application may differ from the Ensemble defaults.
3. Environment variable: **ENS_MODES** variable (see the environment variable documentation below).
4. Ensemble defaults: **UDP**.

2

3.3 Using Deering IP Multicast

The method described above for running the mtalk demo is the best way to first run mtalk because it uses **UDP** for both transports. **UDP** does not use IP multicast communication, which can be a source of problems because of variations in how it is configured at different sites. [**IP multicast is only available when using the Socket library on Unix. It is not currently supported by Ensemble on Windows NT.**] If your machines support Deering IP multicast communication, it is preferable to use **DEERING** transports because you will then not have to run the gossip server with Ensemble applications. You can try out the IP multicast transport by using the command-line arguments. (Note the problems section at the end of this section, however, which describes some of the problems you may have.) This is done by executing the applications with the **-modes** command-line arguments. With IP multicast you no longer need to have a gossip server running. Run the application on the hosts with these arguments (see below for a description of the arguments):

```
% mtalk -modes DEERING
```

To always use IP multicast by default, modify the ENS_MODES environment variable so that it includes DEERING. Also set the ENS_DEERING_PORT environment variable to an unused port number. You will probably wish to add these to your standard shell environment:

```
setenv ENS_DEERING_PORT 1234
setenv ENS_MODES DEERING:UDP
```

3.4 Notes and Problems

See also the problems mentioned in the Ensemble reference manual.

IP Multicast problems : Some problems may occur with IP Multicast. The time-to-live value for multicast messages may be too small in some environments, preventing multicast messages from reaching all members. The TTL value can be adjusted by editing the file **socket/multicasts.c**.

4 Text-based Application Interface

This is the simplest of the Ensemble application interfaces in the sense that it is totally language independent. Read section 2.3 describing the **demo/ensemble** tool. A text-based application can easily be built using this tool. The idea is to run the **demo/ensemble** program as a sub-process of an application, connected via a pair of pipes. The Ensemble sub-process will manage the communication. This way the application be built in just about any programming language, especially languages such as Tcl, Perl, and Python. This method suffers from considerably higher communication overheads than single-process solutions.

5 Ensemble ML Application Interface

[**TODO: add example handlers from mtalk**]

We present a simple interface for building single-group applications. This interface is intended to make small applications easy to build, and to protect users from complications in the internals of the system.

The interface is implemented as a set of callbacks the application provides to Ensemble. The application is notified through these callbacks (in a similar fashion to callbacks with Motif widgets) of events that occur in the system, such as message receipts and membership changes.

The interface for a member of a group is always in one of two states, *blocked* or *unblocked.* While unblocked, only the **recv_send**, **recv_cast**, and **heartbeat** callbacks are enabled. This is the normal state of the system. While block, the application *should* refrain from sending messages. However, it can send messages, causing the system to fail with the notification "sending while blocked".

Messages are sent by returning from these callbacks lists of actions to take. An action is usually a message send: either a **Cast** (group broadcast) or a **Send** (point-to-point message). Thus, messages are delivered by callbacks from Ensemble and further messages are sent by returning values from these callbacks.

5.1 Compilation

Compiling ML applications is easy. You can use **demo/Makefile** as a skeleton for your own applications.

5.2 Interface Definition and Initialization

Below is the full ML interface type definition for the application interface described here. A group member is initialized by creating an interface record which defines a set of callback handlers for the application. This is then passed to one of the Ensemble stack initialization functions exported by **appl/appl.mli**.

```
(* Some type aliases.
 *)
type rank = int
type view  = Endpt.id list
type origin  = rank
type dests  = rank array

type control =
  | Leave
  | Prompt
  | Suspect of rank list

  | XferDone
  | Rekey of bool
  | Protocol of Proto.id
  | Migrate of Addr.set
  | Timeout of Time.t            (* not supported *)

  | Dump
  | Block of bool                (* not for casual use *)
  | No_op

type ('cast_msg,'send_msg) action =
  | Cast of 'cast_msg
  | Send of dests * 'send_msg
  | Send1 of rank * 'send_msg
  | Control of control
```

```
(* APPL_INTF.New.full: The record interface for applications.  An
 * application must define all the following callbacks and
 * put them in a record.
 *)

  type cast_or_send = C | S
  type blocked = U | B

  type 'msg naction = ('msg,'msg) action

  type 'msg handlers =
    flow_block : rank option * bool -> unit ;
    block : unit -> 'msg naction array ;
    heartbeat : Time.t -> 'msg naction array ;
    receive : origin -> blocked -> cast_or_send -> 'msg -> 'msg naction array ;
    disable : unit -> unit


  type 'msg full =
    heartbeat_rate : Time.t ;
    install : View.full -> ('msg naction array) * ('msg handlers) ;
    exit : unit -> unit

}
```

5.3 Actions

Some callbacks allow a (possibly empty) array of actions to be returned. There are 4 different kinds of actions:

Cast(msg) : Causes **msg** to be broadcast to the group.

Send(dests,msg) : Causes **msg** to be sent to a subset of the group specified in **dests**. **dests** is an array of ranks.

Send1(dest,msg) : Same as **Send**, but sends **msg** to a single destination. This is slightly more efficient for single destinations.

Control c : This bundles together all control actions. There are several of these:

- **Leave** : Causes the member to leave the group. There should always be at most one **Leave** action returned in an action array.
- **Prompt** : Ask the system to perform a view-change immediately.
- **XferDone** : Signals that this member has completed its state transfer. If a state transfer layer is in the protocol stack, this will trigger a new non-state transfer view after all members have taken an **XferDone** action.
- **Rekey opt** : Ask the system to rekey itself. This should be done in case the current key may have been compromised, for example, if a previously trusted member should be

expelled. The **opt** parameter describes whether previously constructed pt-2-pt session keys can be used to optimize this operation, or whether this is disallowd. For the casual user, the optimized version (opt = false) should be used.

Protocol(protocol) : Requests a protocol switch. If the stack supports protocol switches, a new view will be triggered.

Dump : Causes some debugging output to be printed by the stack in use. The output depends greatly on the protocol stack.

The rest of the actions are not intended for the casual user, they are either not supported, badly supported, or used by system internals.

5.4 The install callback

Whenever a new view is installed, the application install callback is called. This handler describes several callbacks:

```
type 'msg handlers =
  flow_block : rank option * bool -> unit ;
  block : unit -> 'msg naction array ;
  heartbeat : Time.t -> 'msg naction array ;
  receive : origin -> blocked -> cast_or_send -> 'msg -> 'msg naction array ;
  disable : unit -> unit
```

flow_block source onoff is called whenever there are flow control issues. The **onoff** value describes whether communication on the specific channel can resume, or should be held back momentarily until communication problems are resolved. If the **source** is None, then the problematic channel is multicast, if it is **Some(rank)** then there are issues with the point-to-point connection between this endpoint, and endpoint **rank**.

block () is called to notify the application to stop sending messages, because a view change is pending. It is an error to send messages from now on, until a new view is installed, and **install** will be called again.

heartbeat current_time is regularly called by Ensemble when the application is unblocked. The expected rate of heartbeats is specified through the **heartbeat_rate** field of the interface record. The return values for all of these callbacks is an action array.

receive origin bk cs msg is called when a message has been received. The callback is made with the origin of the message, the current block state (bk), if this is a Cast of Send message (cs) and the message itself.

The install callback is called with the current view state, it returns a set of 5 handlers, and also a set of actions to be performed immediatly. It is wrapped up in a structure bundling the heartbeat rate, exit function (see below), and itself.

5.5 View state

Several callbacks receive as an argument a pair of records with information about the new view. The information is split into two parts, a **View.state** and a **View.local** record. The first contains information that is common to all the members in the view, such as the **view** of the group. The same record is delivered to all members. The second record contains information local to the member that receives it. These fields include the **Endpt.id** of the member and its **rank** in the view. It also contains information that is derived from the **View.state** record, such as **nmembers** with is merely the length of the **view** field.

```
(* VIEW.STATE: a record of information kept about views.
 * This value should be common to all members in a view.
 *)
type state =
  (* Group information.
   *)
  version       : Version.id ; (* version of Ensemble *)
  group : Group.id ; (* name of group *)
  proto_id : Proto.id ; (* id of protocol in use *)
  coord         : rank ; (* initial coordinator *)
  ltime         : ltime ; (* logical time of this view *)
  primary       : primary ; (* primary partition? (only w/some protocols) *)
  groupd        : bool ; (* using groupd server? *)
  xfer_view : bool ; (* is this an XFER view? *)
  key : Security.key ; (* keys in use *)
  prev_ids      : id list ;             (* identifiers for prev. views *)
  params        : Param.tl ; (* parameters of protocols *)
  uptime        : Time.t ; (* time this group started *)

  (* Per-member arrays.
   *)
  view  : t ; (* members in the view *)
  clients : bool Arrayf.t ; (* who are the clients in the group? *)
  address       : Addr.set Arrayf.t ; (* addresses of members *)
  out_of_date   : ltime Arrayf.t ; (* who is out of date *)
  lwe           : Endpt.id Arrayf.t Arrayf.t ; (* for light-weight endpoints *)
  protos        : bool Arrayf.t   (* who is using protos server? *)
```

```
(* VIEW.LOCAL: information about a view that is particular to
 * a member.
 *)
type local =
  endpt         : Endpt.id ; (* endpoint id *)
  addr         : Addr.set ; (* my address *)
  rank          : rank ; (* rank in the view *)
  name : string ; (* my string name *)
  nmembers  : nmembers ; (* # members in view *)
  view_id  : id ; (* unique id of this view *)
  am_coord      : bool ;   (* rank = vs.coord? *)
  falses        : bool Arrayf.t ;       (* all false: used to save space *)
  zeroes        : int Arrayf.t ;        (* all zero: used to save space *)
  loop          : rank Arrayf.t ;       (* ranks in a loop, skipping me *)
  async         : (Group.id * Endpt.id) (* info for finding async *)


(* LOCAL: create local record based view state and endpt.
 *)
val local : debug -> Endpt.id -> state -> local
```

Most of the fields are moderately self-explanatory. If **xfer_view** is true, then this view is only for state transfer and all members should take an **XferDone** action when the state transfer is complete. The view field is defined as **View.t**, which is:

```
(* VIEW.T: an array of endpt id's.
 *)
type t = Endpt.id Arrayf.t
```

5.6 Asynchronous operation

The application can only send messages when handling a callback. Under some circumstances (such as when receiving input from another source), it is necessary to send messages immediately rather than waiting for the next regularly scheduled heartbeat to occur. Call the function **Appl.async** with the group and endpoint of the group. This returns a function that can be called whenever an immediate hearbeat is desired. [**This replaces the previous heartbeat_now callback.**]

```
  let async = Appl.async (group,endpt) in
  async ()
```

5.7 Exit notice

Called when the member has left the group (through a previous **Leave** action). This is the last callback the group member will receive.

```
  exit                  : unit -> unit ;
```

5.8 Properties

The Ensemble **Property** module is used to construct protocols based on desired properties the application wants. You can look at **appl/property.mli** for the various properties that are supported by Ensemble:

```
type id =
  | Agree (* agreed (safe) delivery *)
  | Gmp (* group-membership properties *)
  | Sync (* view synchronization *)
  | Total (* totally ordered messages *)
  | Heal (* partition healing *)
  | Switch (* protocol switching *)
  | Auth (* authentication *)
  | Causal (* causally ordered broadcasts *)
  | Subcast (* subcast pt2pt messages *)
  | Frag (* fragmentation-reassembly *)
  | Debug (* adds debugging layers *)
  | Scale (* scalability *)
  | Xfer (* state transfer *)
  | Cltsvr (* client-server management *)
  | Suspect (* failure detection *)
  | Flow (* flow control *)
  | Migrate (* process migration *)
  | Privacy (* encryption of application data *)
  | Rekey (* support for rekeying the group *)
  | OptRekey (* optimized rekeying protocol *)
  | DiamRekey                           (* Diamond rekey algorithm *)
  | Primary (* primary partition detection *)
  | Local (* local delivery of messages *)
  | Slander (* members share failure suspiciions *)
  | Asym         (* overcome asymmetry *)

    (* The following are not normally used.
     *)
  | Drop (* randomized message dropping *)
  | Pbcast (* Hack: just use pbcast prot. *)
  | Zbcast                              (* Use Zbcast protocol. *)
  | Gcast                               (* Use gcast protocol. *)
  | Dbg                                 (* on-line modification of network topology *)
  | Dbgbatch                            (* batch mode network emulation *)
  | P_pt2ptwp                           (* Use experimental pt2pt flow-control protocol *)
```

Here is a short description of some of the properties:

- Gmp: Group Membership Properties.
- Sync: Synchronizes messages on view changes to ensure view synchrony.
- Total: Broadcast messages are totally ordered in the group.

- Heal: Group partitions are healed.
- Switch: Allows on-the-fly protocol switching.
- Auth: Allows only authenticated and authorized members into the group. Creates secure agreement in the group on a mutual group key. This key is used to sign and verify, using keyed-MD5, all group messages. This protects the group from outisde attack.
- Rekey: Allows rekeying the group.
- Privacy: Encrypts all user messages.
- Causal: Broadcasts are causally ordered.
- Subcast: Point-to-point messages are sent using filtered broadcasts. Guarantees FIFO ordering between broadcasts and point-to-point messages.
- Frag: Message fragmentation. Allows messages of any size to be sent.
- Debug: Inserts a variety of “assertion” protocols that check that other properties are being met.
- Scale: Switches some protocols with more scalable versions.
- Xfer: Causes the state transfer field (**xfer**) of view states to be set.
- Cltsvr: Causes the clients field of view states to be set according to whether members are “clients” or “servers”.
- Suspect: Members watch other members for suspected failures.
- Zbcast: A probabilistic multicast protocol, does not guaranty virtual syncrhony. Has been used for experimental studies. See the Cornell Spinglass system for more details.
- Gcast: A protocol that simulates IP-multicast useing a binary tree of pt-2-pt connections between group members.

The **Property.choose** function selects a protocol stack based on a list of desired properties (you can examine the implementation to see exactly how this is done):

```
(* Create protocol with desired properties.
 *)
val choose : id list -> Proto.id
```

The default properties used for Ensemble applications is **Property.vsync**. This is one of a variety of predefined protocol property lists defined in the **Property** module:

```
let vsync = [Gmp;Sync;Heal;Migrate;Switch;Frag;Suspect;Flow]
let total = vsync @ [Total]
let scale = vsync @ [Scale]
let fifo = [Frag]
```

In order to set the properties used by an application, you would use the following code:

```
(* Choose default view state.
 *)
let vs = Appl.default_info "my-appl" in

(* Select desired properties.
 *)
let properties = [ (* list of properties *) ] in

(* Choose corresponding protocol stack.
 *)
let proto_id = Property.choose properties in

(* Set proto_id of the view state record.
 *)
let vs = View.set vs [Vs_proto_id proto_id] in

(* Configure the application
 *)
Appl.config_new my_interface vs ;
```

As described in the reference manual, each of these protocols are derived by combining a set of protocol layers together to get a full protocol stack with application-level properties. Anyway, here we describe the behavior of the **vsync** protocol stack.

- The first callback a protocol stack receives is an **install** with a singleton view.
- All members in the same partition of a group receive the same **View.state** records (excepting the **rank** field, of course).
- **Send** messages are delivered reliably and in FIFO order. It is an error for a member to send a message to itself.
- **Cast** messages are delivered reliably and in FIFO order. FIFO order for **Cast** messages means that members receive the messages in the order they were sent by the sender. **Cast** messages are usually not delivered to the sender (the primary exceptions are stacks with total-ordering layers in them).
- There is no ordering relationship *between* **Send** and **Cast** messages.
- Messages are delivered in the same view they were sent in (the protocol stack "blocks" so that the protocols can flush all the current messages out of the system before advancing to the next view).
- **Cast** messages are delivered atomically. This means that either all members (excepting the sender) or none will receive a **Cast** message. If the sender of a **Cast** message fails, other members who received the message will retransmit it for the failed member. When there is more than one member in a group, a **Cast** message may be delivered to no members only if the sender fails.

- All members that receive the same consecutive views (they get the same **install upcalls** will have delivered the same set of **Cast** messages between the upcalls (but not necessarily in the same order). Thus views can be considered as synchronization points where all members agree on what has been done so far.

5.9 Initializing Ensemble Applications

This is a description of how simple applications are initialized with Ensemble. The source code presented here is extracted from the **mtalk** demo, which is distributed with Ensemble. The source can be found in **demo/mtalk.ml** which compiles and links with the Ensemble library to form the **demo/mtalk** executable.

An application consists of two parts, initialization and an interface. The initialization involves setting up Ensemble and the communication framework. An interface consists of a set of callback handlers that manage application events that Ensemble generates for messages and membership changes. The initialization code tends to be similar across applications, and the handlers tend to contain most of the application-specific functionality. We present a sample set of initialization code, which can easily be adapted for other simple applications. We do not describe the callback handlers here; they are described in section 5. For specific examples, see **demo/mtalk.ml** and **demo/rand.ml**.

```
let run () =
  (*
   * Parse command line arguments.
   *)
  Arge.parse [
    (*
     * Extra arguments can go here.
     *)
  ] (Arge.badarg name) "mtalk: multiperson talk program" ;

  (*
   * Get default transport and alarm info.
   *)
  let view_state = Appl.default_info "mtalk" in

  let alarm = Alarm.get_hack () in
```

The initialization must do several things, all of which can be contained in a single function, as shown here with the function **run**. First parse the command-line arguments as is done above. In addition to arguments provided by the applicatoin, this parses the standard Ensemble arguments. Then, **default_info** is called. This initializes a **View.state** record (which contains all the information other modules need to initialize your application).

```
  (*
   * Choose a string name for this member.  Usually
   * this is "userlogin@host".
   *)
  let name =
    try
      let host = gethostname () in

      (* Get a prettier name if possible.
       *)
      let host = string_of_inet (inet_of_string host) in
      sprintf "%s@%s" (getlogin ()) host
    with _ -> view_state.name
  in

  (*
   * Initialize the application interface.
   *)
  let interface = intf name alarm in
```

Next we initialize the interface record that contains the application's handlers and which does the actual work of the application. How the interface is initialized is application dependent. For example, **interface** will usually require several arguments. In the **mtalk** application, the interface takes the endpoint identifier of the application and a string name to use for this member of the talk group. Other applications will use different arguments.

```
  (*
   * Initialize the protocol stack, using the interface and
   * view state chosen above.
   *)
  Appl.config_new interface view_state ;
```

The code above initializes the protocol stack. In this case we use the **vsync** protocol properties, which provide FIFO, virtually-synchronous communication and an automatic merging facility for healing partitions. There are several different sets of properties by the **appl/property.mli** module, each of which provides different properties or performance characteristics (for more information about properties, see section 5.8).

```
  (*
   * Enter a main loop
   *)
  Appl.main_loop ()
  (* end of run function *)


(* Run the application, with exception handlers to catch any
 * problems that might occur.
 *)
let _ = Appl.exec ["mtalk"] run
```

The initialization is complete and we enter a main loop. The main loop never returns. The final code calls the **run** function with some standard exception handlers to catch any exceptions that should not, but may, occur.

This is all that is required for initializing simple, single-group Ensemble applications.

6 Using PGP

Ensemble supports the use of PGP for authenticating members of groups. This work is complete, and several papers have been pusblished with our results. We do not guarantee bullet proof security, however, we do not know of any remaining security bugs. All the Ensemble demo applications support the use of PGP, including **mtalk**, **wbml**, and **ensemble**.

These are the instructions for using PGP. Note that PGP is supported for all platforms.

- The **pgp** binary must be in your path. Ensemble executes PGP as a subprocess for authenticating remote members. If you do not yet have a PGP keyring, read the PGP documentation on how to set all this up.
- You must set the **PGPPASS** environment variable to contain your secret key pass phrase. See the PGP documentation for more information.
- **-pgp user** : command line argument. This tells Ensemble what this user's name is for PGP other processes will use this name to select the public key to use for authenticating you.
- **-key sharedkey**: command line argument. This sets the shared key conversation key that Ensemble will use initially. It should be at least 32 characters long.
- **-add_prop Auth**: command line argument. This adds the **Auth** property to the default Ensemble properties. This then causes the **EXCHANGE** protocol to be used in the protocol stack for exchanging shared keys.

Now when you run an application only members that start with the same shared key or who can authenticate each other through PGP will merge into the same group.

If you run into problems, you can access PGP's debugging output through the additional command-line arguments, **-trace PGP**.

7 Native C Ensemble Application Interface (CE)

The C application interface is very similar in design to the ML interface. It is located in directory **ce**. It has been modified from the original ML interface, so as to fit better into the C language (type-system and native data structures).

There are seven callbacks a C application needs to define in order to work with Ensemble. These are:

- `install(env,ls,vs)` : called whenever a new view is installed.
- `exit()` :called when the member leaves.
- `receive_cast(env, origin, num, iovl)` : called with the origin, an iovec array (and its length) whenever a mulicast message arrives.
- `receive_send(env, origin, num, iovl)` : called with the origin, an iovec array (and its length) whenever a point-to-point message arrives.
- `flow_block(env, origin, onoff)` : called whenever there are flow-control problems, and the application should refrain from sending messages until further notice.
- `block(env)` : called whenever a view change is forthcoming. All applications are blocked, the old view is stabilized, cleaned, and way is made for the new view.
- `heartbeat(env, time)` : called every timeout. The timeout is specified in the `jops` structure. Timers are not exact, this callback may be called at inaccurate times, or more often than neccessary. If accuracy is required, the application should check the *time* argument.

The environment argument which is the first argument in all seven callbacks is registered when a *C-application interface* is created.

The types of the callbacks are as follows:

```
typedef int          ce_rank_t ;
typedef int          ce_len_t ;
typedef void        *ce_env_t ;
typedef double       ce_time_t ;

typedef void (*ce_appl_install_t)(ce_env_t, ce_local_state_t*, ce_view_state_t*);

typedef void (*ce_appl_exit_t)(ce_env_t) ;

typedef void (*ce_appl_receive_cast_t)(ce_env_t, ce_rank_t, int, ce_iovec_array_t) ;

typedef void (*ce_appl_receive_send_t)(ce_env_t, ce_rank_t, int, ce_iovec_array_t) ;

typedef void (*ce_appl_flow_block_t)(ce_env_t, ce_rank_t, ce_bool_t) ;

typedef void (*ce_appl_block_t)(ce_env_t) ;

typedef void (*ce_appl_heartbeat_t)(ce_env_t, ce_time_t) ;
```

A `ce_appl_intf_t` is the type of a C application interface (*cappl*). It can be created by the constructor `ce_create_intf`. There is no need for a destructor because Ensemble frees the interface-structure and all related memory after the `exit` callback is invoked. An application interface is opaque, it can be used to create and endpoint, and join a group. It cannot be used to join more than a single group.

```
typedef struct ce_appl_intf_t ce_appl_intf_t ;
```

The constructor takes the above handlers as parameters, as well as an environment variable.

```
ce_appl_intf_t*
ce_create_intf(
    ce_env_t env,
    ce_appl_exit_t exit,
    ce_appl_install_t install,
    ce_appl_flow_block_t flow_block,
    ce_appl_block_t block,
    ce_appl_receive_cast_t cast,
    ce_appl_receive_send_t send,
    ce_appl_heartbeat_t heartbeat
);
```

The initial operation used to initiate a CE application is `ce_Init`. It initializes the internal Ensemble data structures, and processes command line arguments.

```
void ce_Init(int argc, char **argv) ;
```

After a C application completes initialization it should pass control the Ensemble main loop via `ce_Main_loop`.

```
void ce_Main_loop ();
```

In order to Join a group, the `ce_Join` operation should be used.

```
void ce_Join(ce_jops_t *ops, ce_appl_intf_t *c_appl) ;
```

7.1 Group operations

Similarly to the ML interface, the set of supported operations is: Leave, Cast, Send, Send1, Prompt, Suspect, XferDone, Rekey, ChangeProtocol, and ChangeProperties. Messages are arrays of IO-vectors (*iovecs*), or C memory chunks. The application can send and receive iovec-arrays.

Multicast an iovec-array to the group.

```
void ce_Cast(
    ce_appl_intf_t *c_appl,
    int num,
    ce_iovec_array_t iovl
) ;
```

Send a point-to-point message to a set of group members.

```
void ce_Send(
    ce_appl_intf_t *c_appl,
    int num_dests,
    ce_rank_array_t dests,
    int num,
    ce_iovec_array_t iovl
) ;
```

Send a point-to-point message to the specified group member.

```
void ce_Send1(
    ce_appl_intf_t *c_appl,
    ce_rank_t dest,
    int num,
    ce_iovec_array_t iovl
) ;
```

The control actions, are the same as the ML actions.

Leave a group. Following this downcall, `exit` will be called, freeing the cappl.

```
void ce_Leave(ce_appl_intf_t *c_appl) ;
```

Ask for a new View.

```
void ce_Prompt(
    ce_appl_intf_t *c_appl
);
```

Report specified group members as failure-suspected.

```
void ce_Suspect(
    ce_appl_intf_t *c_appl,
    int num,
    ce_rank_array_t suspects
);
```

Inform Ensemble that the state-transfer is complete.

```
void ce_XferDone(
    ce_appl_intf_t *c_appl
) ;
```

Ask the system to rekey.

```
void ce_Rekey(
    ce_appl_intf_t *c_appl
) ;
```

Request a protocol change. The `protocol_name` is a string specifying the exact set of layers to use. The string is a colon separated list of layers, for example: Top:Heal:Switch:Leave:Inter:Intra:Elect:Merge:Sy Vsync:Frag_Abv:Top_appl:Frag:Pt2ptw:Mflow:Pt2pt:Mnak:Bottom

```
void ce_ChangeProtocol(
    ce_appl_intf_t *c_appl,
    char *protocol_name
) ;
```

Request a protocol change, specifying properties. `properties` is a string containing a colon separated list of properties. For example: "Gmp:Sync:Heal:Switch:Frag:Suspect:Flow:Xfer". The system deduces a protocol stack that abides by these properties.

```
void ce_ChangeProperties(
    ce_appl_intf_t *c_appl,
    char *properties
) ;
```

7.2 Integration of other sockets into the main loop

Ensemble works in an event driven fashion, where events can either come from the network or the user. The system runs a loop that is split between (1) waiting for input on incoming sockets using a `select` system call (2) Processing local application send/recv and internal events.

The application hands over control to Ensemble after initialization. The application may wish to wait on its own sockets, e.g., `stdin` (on Unix). To this end, we also support adding, removing, and putting handlers on sockets.

`ce_handler_t` is the type of handler called when there is input to process on a socket.

```
typedef void (*ce_handler_t)(void*);
```

`ce_AddSockRecv` adds a socket to the list Ensemble listens to. When input on the socket occurs, this handler will be invoked on the specified environment variable.

```
void ce_AdddSockRecv(
    CE_SOCKET socket,
    ce_handler_t handler,
    ce_env_t env
);
```

`ce_RmvSockRecv` is called to remove a socket from the list Ensemble listens to.

```
void ce_RmvSockRecv(
    CE_SOCKET socket
);
```

7.3 Memory management

The convention used throughout is that all data-structures passed from C to ML are consumed by ML, and all data-structures passed from ML to C are owned by the C side (hence must be freed). This rule holds for all structures and data apart from the iovec-arrays.

Ensemble does not copy messages from C to the ML heap, rather, it separates C-memory and ML memory completely. Messages are received from the network and read directly into C-buffers. Sent iovecs are fragmented and sent directly on the network. Messages must be buffered until all group members reliably receive them. To this end, a reference counting scheme is used to track iovec liveness. When an iovec's reference count reaches zero, it is freed. In other words, iovec's are owned by Ensemble. They are received either from the user, or the network.

On linux, the type of an iovec is:

```
typedef struct iovec ce_iovec_t ;
typedef ce_iovec_t *ce_iovec_array_t;
```

To get better control of the iovec memory system, the `alloc` and `free` functions can be set by the user. The definitions are in **lib/mm.h**.

These define the types of alloc and free functions.

```
typedef void* (*mm_alloc_t)(int);
typedef void  (*mm_free_t)(char*);
```

The actual functions called to free and allocate iovec's.

```
mm_alloc_t mm_alloc_fun;
mm_free_t mm_free_fun;
```

Use these functions to set `alloc` and `free`. Be careful to do this exactly once at application initialization, before starting Ensemble.

```
void set_alloc_fun(mm_alloc_t f);
void set_free_fun(mm_free_t f);
```

The upshot of this is that when a user sends or casts a message, Ensemble takes over the message body. When a message is delivered to the application, the user may copy it, or perform any read-only operation while in the receive callback. The application may *not* modify a received iovec, or assume it owns it.

7.4 The flat interface

Using iovecs is a little complex for simple applications, therefore, a simplified "flat" interface exists.

The flat_receive callbacks take a C memory chunk, with it's length as arguments. This releases the application from merging together the set of buffers that consist an iovec-array, as well as releasing that array.

```
typedef void (*ce_appl_flat_receive_cast_t)(ce_env_t, ce_rank_t, ce_len_t, ce_data_t) ;

typedef void (*ce_appl_flat_receive_send_t)(ce_env_t, ce_rank_t, ce_len_t, ce_data_t) ;
```

Create a standard application interface using flat receive callbacks.

```
ce_appl_intf_t*
ce_create_flat_intf(
    ce_env_t env,
    ce_appl_exit_t exit,
    ce_appl_install_t install,
    ce_appl_flow_block_t flow_block,
    ce_appl_block_t block,
    ce_appl_flat_receive_cast_t cast,
    ce_appl_flat_receive_send_t send,
    ce_appl_heartbeat_t heartbeat
);
```

Cast and Send operations that work with buffers instead of iovec-arrays.

```
void ce_flat_Cast(
    ce_appl_intf_t *c_appl,
    ce_len_t len,
    ce_data_t buf
) ;

void ce_flat_Send(
    ce_appl_intf_t *c_appl,
    int num_dests,
    ce_rank_array_t dests,
    ce_len_t len,
    ce_data_t buf
) ;

void ce_flat_Send1(
    ce_appl_intf_t *c_appl,
    ce_rank_t dest,
    ce_len_t len,
    ce_data_t buf
) ;
```

7.5 An example

This section shows how to use the CE interface to write applications. We walk through the **ce/ce_mtalk.c** demo program.

ce/ce_mtalk.c, similarly to **demo/mtalk.ml**, is a multi-person talk program. Messages are read from the user via `stdin`, and multicasted to the network.

`state_t` is the state structure used by the program. It is the environment variable registered in the C-interface. The state contains the current view information, a pointer to its cappl, and a flag indicating if we are blocked.

```
typedef struct state_t {
  ce_local_state_t *ls;
  ce_view_state_t *vs;
  ce_appl_intf_t *intf ;
  int blocked;
} state_t;
```

A helper function to multicast a message if we are not blocked. We use the flat interface, to save the messy handling of iovec's.

```
void cast(state_t *s, char *msg){
  if (s->blocked == 0)
    ce_flat_Cast(s->intf, strlen(msg), msg);
}
```

A handler for stdin. This callback is called whenever there is input on the socket. The handler multicasts any message the user types on the screen. Be careful not to send messages if we are blocked.

```
void stdin_handler(void *env) {
  state_t *s = (state_t*)env;
  char buf[100], *tmp;
  int len ;

  fgets(buf, 100, stdin);
  len = strlen(buf);
  if (len>=100)
    /* string too long, dumping it.
     */
    return;

  tmp = ce_copy_string(buf);
  TRACE2("Read %s:", tmp);
  cast(s, tmp);
}
```

There is nothing special to do if we leave the group, the application essentially halts.

```
void main_exit(void *env)
```

When a new view arrives, update the environment structure. Do not forget to free the old view structure.

```
void main_install(void *env, ce_local_state_t *ls, ce_view_state_t *vs) {
  state_t *s = (state_t*) env;

  ce_view_full_free(s->ls,s->vs);
  s->ls = ls;
  s->vs = vs;
  s->blocked =0;
  printf("%s nmembers=%d", ls->endpt, ls->nmembers);
}
```

Ignore flow control problems. We are not suppose to have any of these, we are very low bandwidth.

```
void main_flow_block(void *env, ce_rank_t rank, ce_bool_t onoff)
```

Mark our blocked flag.

```
void main_block(void *env) {
  state_t *s = (state_t*) env;

  s->blocked=1;
}
```

Print out any message that we receive. Be careful not to free the received message.

```
void main_recv_cast(void *env, int rank, ce_len_t len, char *msg) {
  state_t *s = (state_t*) env;

  printf("recv_cast <- %d msg=%s", rank, msg);
}
```

Ignore send messages, we are not supposed to get any of these.

```
void main_recv_send(void *env, int rank, ce_len_t len, char *msg) {
}
```

Ignore heartbeats.

```
void main_heartbeat(void *env, double time) { }
```

Create a join options structure, and join the group "ce_mtalk". Use a regular virtually-synchronous stack. Put a handler on `stdin` such that whenever there is input, it will be called.

There is no need to set the transport in the join-options structure, the system uses the environment variable ENS_MODES in this case.

```
void join() {
  ce_jops_t *jops;
  ce_appl_intf_t *main_intf;
  state_t *s;

  /* The rest of the fields should be zero. The
   * conversion code should be able to handle this.
   */
  jops = record_create(ce_jops_t*, jops);
  record_clear(jops);
  jops->hrtbt_rate=10.0;
  //  jops->transports = ce_copy_string("UDP");
  jops->group_name = ce_copy_string("ce_mtalk");
  jops->properties = ce_copy_string(CE_DEFAULT_PROPERTIES);
  jops->use_properties = 1;

  s = (state_t*) record_create(state_t*, s);
  record_clear(s);

  main_intf = ce_create_flat_intf(s,
main_exit, main_install, main_flow_block,
main_block, main_recv_cast, main_recv_send,

     main_heartbeat);

  s->intf= main_intf;
  ce_Join (&jops, main_intf);

  ce_AddSockRecv(0, stdin_handler, s);
}
```

The main entry point, initialize the ML side, process command line arguments, join the *ce_mtalk* group, and turn control over to the Ensemble event loop.

```
int main(int argc, char **argv) {

  ce_Init(argc, argv); /* Call Arge.parse, and appl_process_args */

  join();

  ce_Main_loop ();
  return 0;
}
```

7.6 Outboard mode

It is possible to run any CE application through a remote Ensemble server. Such a configuration is called an “outboard” configuration. The idea is to run a daemon on the local host that listens to

TCP connections on a specific port, the daemon provides Ensemble services to connected clients. Such services include joining/leaving groups, and sending/receiving multicast and point-to-point messages on these groups.

A CE application can be configured to run in outboard mode by linking with the `libceo` library (suffix `.a` on Unix, `.lib` on WIN32). The user must then make sure that the Ensemble daemon is running, simply run the ce_outboard executable.

Using a daemon configuration has several benefits as well as some drawbacks. The advantages are:

- The library to link with is orders of a magnitude smaller than the full (inboard) Ensemble library.
- The user-process is completely separated from the Ensemble server. This allows better debugging, and also facilitates writing simple interfaces to other languages (e.g., Java, Ada, ...).

The disadvantage is performance loss. Each message now has to travel through a socket and another process before being sent on the network; vice-versa for received messages. This may outweigh the benefits of simple client code, and a minimal sized library.

The current port used by the outboard mode is 5002. This is configurable by running `ce_outboard` with the command line argument `-tcp_port <port_num>`, and modifying the `OUTBOARD_TCP_PORT` parameter in `ce/ce_outboard_comm.h`.

Care was taken to optimize memory consumption. Messages are sent zero-copy from the client, and they are copied once only into the server's buffers. A sent io-vector is consumed by the send function. Received messages are allocated at the client's buffers and handed to the application. After the application's receive callback, io-vectors are released. It was possible at this point to allow the application to take control of the io-vector, yet we chose to conform with the memory convections of the inboard mode.

7.7 Thread-safety

A thread-safe version of the library is also provided, it exports the exact same interface as the basic library. To use it link with `libce_mt.so`, or `libceo_mt.so`. For WIN32 systems link with `.lib` instead. The thread-safe library requires the application to synchronize its threads so they will not perform actions (send, cast, prompt, etc.) on a group while it is stabilizing. There are several thread-safe applications under the `ce` directory: `ce_rand_mt.c, ce_perf_mt.c`, and `ce_mtalk_mt.c`. These applications use a lock to ensure that sensitive group-state is accessed safely. Threads atomically check group-state before performing an Ensemble action.

The thread-safe library is designed as a wrapper around the basic library. A single thread runs both Ensemble main-loop and application callback handlers; this thread is known as the *Ensemble thread*. Other threads are refered to as *user-threads*. When a user-thread performs an action outside of a handler, the action is stored in a pending queue. A byte is sent through a socket to the Ensemble thread, notifying it that there is pending work to do. Asynchronously, the Ensemble thread "wakes up", consumes the queue, and performs all pending actions. Any actions invoked in the interim will also be stored in the pending queue; to be consumed along with the rest.

Any action invoked from within a callback is performed directly when the callback is completed and control returns to Ensemble.

Since a single thread performs the Ensemble main-loop as well as all user callbacks, callbacks must be short. Long-term computations should **not** be performed in the context of a callback.

There are three sensitive periods in which issuing Ensemble actions is not allowed, these are when *joining*, *leaving*, and *blocking*. A group is in:

joining state: between `ce_Join` and the first `install` callback.

leaving state: between `ce_Leave` and the `exit` callback.

blocking state: between the `block` callback and the succeeding `install` callback.

An example of a simple multi-threaded application is provided in `ce/ce_mtalk_mt.c`.

The overhead of adding thread-safety is 10% in the worst case, and normally much less than that. This should be acceptable for most applications.

7.8 A multi-threaded multi-person chat program

This program is a multi-threaded version of `ce_mtalk.c` Here, we walk through it and explain the interface and how to use it.

Include the system-independent thread header file, so we'll be able to use locks.

```
#include "ce_trace.h"
#include "ce.h"
#include "ce_threads.h"
#include <stdio.h>
#include <memory.h>
#include <malloc.h>
```

The `NAME` variable is used for internal tracing purposes of CE. There is no need to set it for standard user programs.

```
#define NAME "CE_MTALK_MT"
```

Apart for standard view state, the state structure keeps track of the current status of the group: blocked, joining, or leaving.

```
typedef struct state_t {
    ce_local_state_t *ls;
    ce_view_state_t *vs;
    ce_appl_intf_t *intf ;
    int blocked;
    int joining;
    int leaving;
    ce_lck_t *mutex;
} state_t;
```

Although we must define these callbacks, they do nothing in this program.

```
void main_exit(void *env)
{}

void
main_flow_block(void *env, ce_rank_t rank, ce_bool_t onoff)
{}

void
main_recv_send(void *env, int rank, ce_len_t len, char *msg)
{}

void
main_heartbeat(void *env, double time)
{}
```

`main_install` updates the view state. A lock must be taken to protect view state, as other threads may concurrently read the state.

```
void
main_install(void *env, ce_local_state_t *ls, ce_view_state_t *vs)
{
    state_t *s = (state_t*) env;

    ce_lck_Lock(s->mutex); {
        ce_view_full_free(s->ls,s->vs);
        s->ls = ls;
        s->vs = vs;
        s->blocked =0;
        s->joining =0;

        printf("%s nmembers=%d", ls->endpt, ls->nmembers);
        TRACE2("main_install",s->ls->endpt);
    } ce_lck_Unlock(s->mutex);
}
```

The group is blocked, lock the state structure, and update the blocked flag. This notifies other threads not to attempt sending messages until the upcoming install callback. A lock must be taken to protect view state, as other threads may read it.

```
void
main_block(void *env)
{
    state_t *s = (state_t*) env;

    ce_lck_Lock(s->mutex); {
        s->blocked=1;
   } ce_lck_Unlock(s->mutex);
}
```

Received a message, print who sent it and its content.

```
void
main_recv_cast(void *env, int rank, ce_len_t len, char *msg)
{
    printf("%d -> msg=%s", rank, msg); fflush(stdout);
}
```

`get_input` is a non-terminating function performed by the user-thread of this program. In an infinite loop, read a line from stdin, and multicast it to the group. Prior to sending, check that the group is not blocked/joining/leaving. Status flags are shared information, and may be updated concurrently by an `install` or `block` callback. Hence, a lock is taken to protect access to the flags.

```
void
get_input(void *env)
{
    state_t *s = (state_t*)env;
    char buf[100], *msg;
    int len ;

    while (1) {
        TRACE("stdin_handler");
        fgets(buf, 100, stdin);
        len = strlen(buf);
        if (len>=100)
            /* string too long, dumping it.
             */
            return;

        msg = ce_copy_string(buf);
        TRACE2("Read: ", msg);

        ce_lck_Lock(s->mutex); {
            if (s->joining || s->leaving || s->blocked)
                printf("Cannot send while group is joining/leaving/blocked");
            else {
                ce_flat_Cast(s->intf, strlen(msg), msg);
            }
        } ce_lck_Unlock(s->mutex);
    }
}
```

Initialize the state structure, and join the “ce_mtalk” Ensemble group. Take care to initialize the lock, and set the joining flag. The flag will be unset, allowing sending messages, in the first install callback.

```
state_t *
join(void)
{
    ce_jops_t *jops;
    ce_appl_intf_t *main_intf;
    state_t *s;

    /* The rest of the fields should be zero. The
     * conversion code should be able to handle this.
     */
    jops = record_create(ce_jops_t*, jops);
    record_clear(jops);
    jops->hrtbt_rate=10.0;
    jops->transports = ce_copy_string("DEERING");
    jops->group_name = ce_copy_string("ce_mtalk");
    jops->properties = ce_copy_string(CE_DEFAULT_PROPERTIES);
    jops->use_properties = 1;

    s = (state_t*) record_create(state_t*, s);
    record_clear(s);

    main_intf = ce_create_flat_intf(s,
                main_exit, main_install, main_flow_block,
                main_block, main_recv_cast, main_recv_send,
                main_heartbeat);

    s->intf= main_intf;
    s->mutex = ce_lck_Create();
    s->joining = 1;
    ce_Join (jops, main_intf);
    return s;
}
```

Initialize Ensemble, start the reader thread, and go to sleep.

```
int
main(int argc, char **argv)
{
    state_t *s;

    ce_Init(argc, argv); /* Call Arge.parse, and appl_process_args */

    /* Join the group
     */
    s = join();

    /* Create a thread to read input from the user.
     */
    ce_thread_Create(get_input, s, 10000);

    ce_Main_loop ();
    return 0;
}
```

7.9 Notes

Of the four transports supported by Ensemble : NETSIM, UDP, TCP, and DEERING, NETSIM is not supported for the thread-safe library. A socket is used internally, and NETSIM does not allow any external communication. Hence, it is unsupported.

8 Horus Object Tools (HOT)

HOT, the Horus Object Tools, forms the Ensemble C interface. This chapter describes its interfaces, and how to use it.

8.1 Introduction

The *Horus Object Tools* (HOT) is a toolkit developed for the Horus group communication system. Horus is the predecessor of Ensemble, developed at Cornell university in the years 1990-1995.

HOT was ported to Ensemble, and it now forms its C interface. On top of HOT, the maestro toolkit provides a C++ class hierarchy. This document walks through the major HOT interface file - `hot_ens.h`.

8.2 The major interface

File `hot_ens.h` is located in the `include` subdirectory of hot.

```
#define HOT_ENS_DEFAULT_TRANSPORT "UDP"
```

The `HOT_ENS_DEFAULT_TRANSPORT` parameter designates which transport the system is to use. The major options are UDP, TCP, or IP-multicast (DEERING). The default is UDP.

```
#if 0
#define HOT_ENS_DEFAULT_PROTOCOL
      "Top:Heal:Switch:Leave:Inter:Intra:Elect:Merge:Sync:Suspect:"
      "Top_appl:Pt2pt:Frag:Stable:Mnak:Bottom"
#else
#define HOT_ENS_DEFAULT_PROTOCOL
   "BOTTOM:MNAK:PT2PT:CHK_FIFO:CHK_SYNC:"
   "TOP_APPL:STABLE:VSYNC:SYNC:"
   "ELECT:INTRA:INTER:LEAVE:SUSPECT:"
   "HEAL:TOP"
#endif
```

The `HOT_ENS_DEFAULT_PROTOCOL` parameter designates which Ensemble protocol stack to use. There are many options, one may choose to use state-transfer, total-ordering, security, probabilistic protocols, debugging layers and more. This parameter is used, only if the next parameter `HOT_ENS_DEFAULT_PROPERTI` is not used.

```
#define HOT_ENS_DEFAULT_PROPERTIES "Gmp:Sync:Heal:Switch:Frag:Suspect:Flow"
#define HOT_ENS_DEFAULT_HEARTBEAT_RATE 5000
#define HOT_ENS_DEFAULT_GROUP_NAME "<Ensemble_Default_Group>"
```

Since Ensemble supports about 50 layers, and many more layer combinations, simply requesting a layer combination is too low-level. A higher level interface is provided by the `PROPERTIES` parameter. Each property describes a set of layers to use, and a set of properties specifies a layer combination. The default is to use:

Property	description
Gmp	group membership
Sync	Virtual Synchrony
Switch	layer switching on the fly
Frag	message fragmentation
Suspect	Failure suspicion
Flow	Flow control

```
/* Group context descriptor.
 */
typedef struct hot_gctx *hot_gctx_t ;
```

It is possible to join several groups in a HOT application.

```
/* Endpoint id.
 */
typedef struct {
  char name[HOT_ENDP_MAX_NAME_SIZE] ;
} hot_endpt_t ;
```

The type of an endpoint.

```
#define HOT_ENS_MAX_PROTO_NAME_LENGTH 256
#define HOT_ENS_MAX_PARAMS_LENGTH 256
#define HOT_ENS_MAX_PRINCIPAL_NAME_LENGTH 64
#define HOT_ENS_MAX_KEY_LEGNTH     40
#define HOT_ENS_MAX_PROPERTIES_LENGTH 256
#define HOT_ENS_MAX_VERSION_LENGTH 128
#define HOT_ENS_MAX_GROUP_NAME_LENGTH 128
#define HOT_ENS_MAX_IO_NAME_LENGTH 128

typedef struct {
  int ltime ;
  hot_endpt_t coord ;
} hot_view_id ;

typedef unsigned hot_rank_t ;
typedef int hot_bool_t ;
```

Some constants, the view logical time, a member rank, and the boolean type. A *view logical time (lt)* is an identifier that is attached to a *view*. A *view* is the current membership. As group membership changes over time, so do group views. An *lt* is a logical time that allows comparing two views and determining which view is older.

```
/* View state.
 */
typedef struct {
  char version[HOT_ENS_MAX_VERSION_LENGTH] ;
  char group_name[HOT_ENS_MAX_GROUP_NAME_LENGTH] ;
  hot_endpt_t *members ;
  hot_rank_t nmembers ;
  hot_rank_t rank ;
  char protocol[HOT_ENS_MAX_PROTO_NAME_LENGTH] ;
  hot_bool_t groupd ;
  hot_view_id view_id ;
  char params[HOT_ENS_MAX_PARAMS_LENGTH] ;
  hot_bool_t xfer_view ;
  hot_bool_t primary ;
  hot_bool_t *clients;
  char key[HOT_ENS_MAX_KEY_LEGNTH];
} hot_view_state_t ;
```

The view data structure detailing view information. For example: the Ensemble version number, group name, a list of members, etc.

8.3 Application callback types

This section is dedicated to a list of callbacks that a HOT application must provide. These functions are called by Ensemble whenever certain events occur. For example, the view callback is invoked whenever a view change occurs.

```
typedef void (*hot_ens_ReceiveCast_cback)
    (hot_gctx_t gctx, void *env, hot_endpt_t *origin, hot_msg_t msg) ;
```

A multicast message has been received. The parameters are:

gctx: group context.

env: an context parameter.

origin: the origin of the message.

msg: the received message.

```
typedef void (*hot_ens_ReceiveSend_cback)
    (hot_gctx_t gctx, void *env, hot_endpt_t *origin, hot_msg_t msg) ;
```

A point-to-point message has been received, as above.

```
typedef void (*hot_ens_AcceptedView_cback)
    (hot_gctx_t gctx, void *env, hot_view_state_t *view_state) ;
```

A new view has been accepted.

```
typedef void (*hot_ens_Heartbeat_cback)
    (hot_gctx_t gctx, void *env, unsigned rate) ;
```

Got a heartbeat event (invoked periodically). Current time (in milliseconds) is specified.

```
typedef void (*hot_ens_Block_cback)
  (hot_gctx_t gctx, void *env) ;
```

The group is about to block for a view change

```
typedef void (*hot_ens_Exit_cback)
  (hot_gctx_t gctx, void *env) ;
```

The member has left the group.

8.4 Structures and join options

```
/* Application callback configuration.
 */
typedef struct {
  hot_ens_ReceiveCast_cback receive_cast ;
  hot_ens_ReceiveSend_cback receive_send ;
  hot_ens_AcceptedView_cback accepted_view ;
  hot_ens_Heartbeat_cback heartbeat ;
  hot_ens_Block_cback block ;
  hot_ens_Exit_cback exit ;
} hot_ens_cbacks_t ;
```

This is a structure that must be defined by any HOT application. It is passed as an argument to HOT, in the join options (see below).

```
typedef struct {
    unsigned heartbeat_rate; /* Rate of heartbeat upcalls, in millisec. */
    char transports[HOT_ENS_MAX_TRANSPORTS_LENGTH]; /* List of transports */
    char group_name[HOT_ENS_MAX_GROUP_NAME_LENGTH]; /* ASCII name of the group */
    char protocol[HOT_ENS_MAX_PROTO_NAME_LENGTH]; /* The protocol stack */
    char properties[HOT_ENS_MAX_PROPERTIES_LENGTH]; /* Protocol properties */
    hot_bool_t use_properties; /* Set if properties are to be used */
    char params[HOT_ENS_MAX_PARAMS_LENGTH]; /* Protocol parameters */
    hot_bool_t groupd; /* Set if group daemon is in use */
    hot_ens_cbacks_t conf;
    void *env;
    char **argv;
    hot_bool_t debug;
    hot_bool_t client;
    char outboard[HOT_ENS_MAX_IO_NAME_LENGTH]; /* Outboard module to use */

    /* Normally, Ensemble generates a unique endpoint name for each
     * group an application joins (this is what happens if you leave
     * 'endpt' unmodified).  The application can optionally provide
     * its own endpoint name.  It can, for instance, reuse an
     * endpoint name generated by Ensemble for another group (the
     * same endpoint name can be used to join any number of groups).
     * The application can even generate an endpoint on its own.
     * Such names should be unique.  It is best if they contain only
     * printable characters and do not contain spaces because
     * Ensemble my print them out in debugging or error messages.
     * (The names generated by Ensemble fit these characteristics.)
     * See Endpt.extern in ensemble/type/endpt.mli for more
     * information.
     */
    hot_endpt_t endpt ;

    /* [OR] Security options.
     */
    char princ[HOT_ENS_MAX_PRINCIPAL_NAME_LENGTH]; /* Principal name */
    char key[HOT_ENS_MAX_KEY_LEGNTH];             /* key length */

    hot_bool_t secure;                      /* Secure group. */
} hot_ens_JoinOps_t;
```

A list of join options. This includes the transport to be used, the heartbeat rate, the group name, a parameter list, a list of callbacks, an environment variable (used in the callbacks), and more.

The outboard parameter designates whether to use the outboard or not. There are two modes in which HOT can be run, a standard *inboard* mode, and a debugging *outboard* mode. In the inboard mode the HOT C code is directly linked with the Ensemble ML code. In order to allow separately debugging the C code without complications of a shared heap between ML and C objects the outboard mode is provided. In this mode, the HOT C code communicates with Ensemble thorough a TCP/IP socket. The `outboard` parameter describes the path to the outboard executable. This executable is created by the HOT makefile, and it's standard location is in the demo subdirectory.

```
typedef enum {
  HOT_ENS_MSG_SEND_UNSPECIFIED_VIEW,
  HOT_ENS_MSG_SEND_CURRENT_VIEW,
  HOT_ENS_MSG_SEND_NEXT_VIEW
} hot_ens_MsgSendView ;
```

Messages can be sent in the current view, in the next view, etc.

8.5 Application calls

This section describes a list of calls usable directly from the application.

```
hot_err_t hot_ens_Start(
  char **argv
) ;
```

Start the Ensemble thread. The current design is that Ensemble is started in one thread, and HOT code and application are run in another thread. This allows running several application threads using Ensemble in a thread-safe manner.

```
void hot_ens_InitJoinOps(
  hot_ens_JoinOps_t *ops
) ;
```

Initialize an options structure.

```
hot_err_t hot_ens_Join(
  hot_ens_JoinOps_t *ops,
  hot_gctx_t *gctxp /*OUT*/
) ;
```

Join a group. On success, returns group context in `*gctxp`.

```
hot_err_t hot_ens_Leave(hot_gctx_t gctx) ;
```

Leave a group. After this downcall, the context becomes invalid.

```
hot_err_t hot_ens_Cast(
  hot_gctx_t g,
  hot_msg_t msg,
  hot_ens_MsgSendView *send_view /*OUT*/
) ;
```

Send a multicast message to the group. The view in which the message is going to be sent is returned in send_view, if send_view is not NULL. This allows sending a message in the next view.

```
hot_err_t hot_ens_Send(
        hot_gctx_t g,
   hot_endpt_t *dest,
   hot_msg_t msg,
   hot_ens_MsgSendView *send_view /*OUT*/
) ;
```

Send a point-to-point message to the specified group member. The view in which the message is going to be sent is returned in send_view, if send_view is not NULL. This allows sending a message in the next view.

```
hot_err_t hot_ens_Suspect(
   hot_gctx_t gctx,
   hot_endpt_t *suspects,
   int nsuspects
) ;
```

Report specified group members as failure-suspected.

```
hot_err_t hot_ens_XferDone(
   hot_gctx_t gctx
) ;
```

Inform Ensemble that the state-transfer is complete. This callback should be used when the state-transfer layer, XFER, is in the protocol stack. This layer creates a special state-transfer view after a view-change when new members arrive. In this state-transfer view, the application can perform arbitrary communication. Once all application in the view perform an XferDone downcall, the view is terminated, and a new, regular view is installed.

```
hot_err_t hot_ens_ChangeProtocol(
  hot_gctx_t gctx,
  char *protocol_name
) ;
```

Request a protocol change.

```
hot_err_t hot_ens_ChangeProperties(
  hot_gctx_t gctx,
  char *properties
) ;
```

Request a protocol change (specify properties).

```
hot_err_t hot_ens_RequestNewView(
  hot_gctx_t gctx
) ;
```

Request a new view to be installed.

```
hot_err_t hot_ens_Rekey(
  hot_gctx_t gctx
) ;
```

Request a rekey operation.

```
hot_err_t hot_ens_MLPrintOverride(
  void (*handler)(char *msg)
) ;

hot_err_t hot_ens_MLUncaughtException(
  void (*handler)(char *info)
) ;
```

These are functions used to define an action taken when ML throws an exception, or prints out a value. These functions should not be tampered with by the casual user.

Acknowledgements

Thanks to Greg Sharp for comments on previous versions of this document.