

Ensemble Tutorial

Mark Hayden, Ohad Rodeh

Copyright © 1997 Cornell University, 2000 Hebrew University, 2002 IBM Israel Science and Technology

April 11, 2004

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 client application interface.
- The Server (OCaml) Ensemble application interface.

Contents

1	Introduction	4
2	Quick Installation	5
2.1	Compiling	5
2.2	Configuration Variables	5
2.3	Executing Applications	5
I	The Server	6
3	The Programs	7
3.1	Ensembled: the ensemble daemon	7
3.2	Mtalk: Multi-person Talk	7
3.3	Gossip: Group Locator Service	7
3.4	Groupd: Membership Service (formerly called Domain)	7
3.5	Perf: Performance Tests	8
3.6	Rand: Virtual Synchrony Debugging Tool	9
3.7	Fifo: Fifo Communication Debugging Tool	9
3.8	Armadillo: testing Ensemble security extensions	9
3.9	Socketest	10
4	Configuration	11
4.1	Command-line Arguments and Environment Variables	11
4.2	Gossip service configuration	12
4.3	Transports	13
4.4	Using Deering IP Multicast	14
4.5	Notes and Problems	14
5	Server ML Application Interface	15
5.1	Compilation	15
5.2	Interface Definition and Initialization	15
5.3	Actions	17
5.4	The install callback	18
5.5	View state	18
5.6	Asynchronous operation	20
5.7	Exit notice	20
5.8	Properties	21
5.9	Initializing Ensemble Applications	24
6	Using PGP	27
7	Heterogeneous Transports	28
7.1	Code walk-through	28
7.2	Design of the routers	30
II	The Client	32

8	Java Application Interface	33
8.1	The client state-machine	35
8.2	Locking	35
8.3	The View structure	35
8.4	Join options	36
8.5	Limitations	37
8.6	Code examples	38
9	C Application Interface	39

1 Introduction

This documentation assumes that the reader has some familiarity with group communication. There are many texts that describe how to use and build group-communication system.

Ensemble is structured as a client-server system with a server providing group-communication services through a socket based interface. Clients can connect to the server and send/receive reliable point-to-point and multicast messages. There should be one server running on a host, and clients should be located on the same host. This allows using insecure communication for client-server traffic. The server is written (mostly) in the OCaml programming language, the client is a small library that has implementations in several languages. At the time of writing there are clients in C and Java.

Previous versions of the system did not distinguish between client and server. The client was implemented with an internal server. This provides good performance. However, since the server is written in ML, in order to link with a C program written by a user the foreign-language interface of ML needs to be used. This causes very difficult portability issues. As of release 2.00 we decided to separate client from server; this should improve portability at the expense of performance.

2 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.

2.1 Compiling

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

2.2 Configuration Variables

Detailed information is given in Section 4.1 for initializing environment variables. We assume that you will be using IP-multicast as a communication substrate this means no configuration is necessary. However, if multicast is not supported by your system you'll be using the gossip server for processes to locate each other, see Section 4.1 for more information.

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_CONFIG_FILE=/etc/ensemble.conf
```

If you're using a win32 system you will need to use the native environment-setting tool (`start` → `setting` → `control-panel` → `system` → `advanced` → `environment-variables`) which provides similar functionality.

2.3 Executing Applications

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

```
% mtalk
...
```

Applications should merge together and form a group.

Mtalk is a *server-based* program, this means that it is really on the server-side, not a client application. Mtalk is useful for testing if servers on different machines can talk to each other and merge to form groups. We discourage users from writing server-side programs, it is easier and safer to write client-side programs.

Part I

The Server

This chapter describes how to build server-side programs. The reason users should be wary of writing such programs is that the server operates in a soft real-time environment. The server is written in the OCaml programming language, a single thread of execution is used. To improve performance bulk-data for user messages is not allocated on the ML heap, which is garbage collection, it is allocated on large chunks of memory allocated with `malloc`. Bulk-data extents are also called io-vectors and the memory used to hold them is also called iovec-memory. To reduce server memory foot-print the size of iovec-memory is limited, at the time of writing we are using 6 mega-bytes. Since memory is limited memory-allocation can fail. The server handles this with flow-control protocols limiting the amount of incoming messages to fit the amount of available iovec-memory. In order to maintain responsiveness to incoming messages pure CPU processing (such as search on a database) should be limited. If the messaging engine does not receive sufficient CPU every, say, 100 milliseconds then performance is going to suffer dramatically.

The casual user will be better served by the chapter on writing client programs that do not suffer from these limitations. If, however, you are undaunted then this chapter is for you.

3 The Programs

Of the programs described here only `mtalk`, `gossip`, `groupd`, and `ensembled`, are normal programs. The rest: `perf`, `rand`, `fifo`, `socktest`, and `armadillo`, are internal system tests not for use by the casual user.

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.

3.1 Ensembled: the ensemble daemon

This is the ensemble daemon. By running a single instance of this program on a host all clients on that host will be able to use Ensemble services.

3.2 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.

3.3 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 “gossiping” 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 (**DEERING** is the default), it is necessary for a **gossip** process to be running somewhere in the system.

3.4 Group: 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.

3.5 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.**]

3.6 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.

3.7 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.

3.8 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 initial 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 rekeying 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.

3.9 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 loopback. 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.

4 Configuration

4.1 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/arg.ml** for the authoritative list of the configuration parameters. Some are listed below as command-line options. The corresponding configuration variable for **-group_name** (for example) is **ENS_GROUP_NAME** (the name is capitalized and the '-' is replaced 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.
- host_ip IP address**: set the default IP address of the host for Ensemble applications. This allows overriding the default IP-address of the machine, a useful option if the machine is connected to several network interface cards (NICs) and has several IP addresses.
- 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 3.4. 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.
- 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.

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.

Prior to version 1.42 all configuration variables were defined in the environment. This meant that the environment-variable system was tainted with the various Ensemble configuration options. Starting with version 1.42 configuration is handled using a configuration-file which is by default `$USER/ensemble.conf`. To set this to a different file use the **ENS_CONFIG_FILE** environment variable.

Each line in the configuration file is either (1) a comment starting with `#` (2) an empty line (3) a value binding where *ensemble_variable* = *value*. For example:

```
# Ensemble ID
ENS_ID=orodeh

# The port used by Ensemble for transport
ENS_PORT=6790
```

The first operation an Ensemble process performs is to open and parse the configuration file. Note that an option set in the configuration file takes effect for all your Ensemble processes.

4.2 Gossip service configuration

Not all platforms support IP-multicast. Furthermore, if you compile Ensemble without the default native socket library you'll be using OCaml's networking library which, up to version 3.06, did not support multicast. If you fall under any of these categories you'll need to use the gossip service to allow applications 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 configuration variables as follows (in the configuration file):

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

Applications will require a gossip server process to be running in order to contact each other. Before executing an application, a gossip server must be started on 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
...
```

4.3 Transports

[If you are only using regular IP-multicast 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
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.

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: **DEERING**.

4.4 Using Deering IP Multicast

Deering IP Multicast is the default configuration. If your machines support IP multicast communication, it is preferable to use **DEERING** transports because you will then not have to run the gossip server with Ensemble applications. IP multicast is only available when using the Socket library. It is not currently supported by the native OCaml library. The configuration parameters pertaining to Deering are **ENS_MODES** and **ENS_DEERING_PORT**. To modify their default settings use:

```
ENS_DEERING_PORT=1234
ENS_MODES=DEERING
```

You can try out the IP multicast transport by using the command-line arguments. For example:

```
% mtalk -modes DEERING
```

4.5 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/s/PLATFORM/multicasts.c**.

Variation in site configuraiton : IP-multicast may be misconfigured or disabled for security reasons on your site. You’ll need to use the gossip service in this case. To do so set:

```
ENS_MODES=UDP
```

This will tell the system to use UDP instead of DEERING.

5 Server 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 disallowed. 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 immediately. 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** which 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.t1 ; (* 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 heartbeat 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 *)
| Primary    (* primary partition detection *)
| Local      (* local delivery of messages *)
| Slander    (* members share failure suspicions *)
| Asym       (* overcome asymmetry *)

(* The following are not normally used.
   *)
| Drop       (* randomized message dropping *)
| Pbcst      (* Hack: just use pbcst prot. *)
| Zbcst      (* Use Zbcst 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 outside 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.
- Zbcst: A probabilistic multicast protocol, does not guaranty virtual synchrony. Has been used for experimental studies. See the Cornell Spinglass system for more details.
- Gcast: A protocol that simulates IP-multicast using 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 server applications.

6 Using PGP

The Ensemble server supports the use of PGP for authenticating members of groups. This work is complete, and several papers have been published 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, for example, **mtalk**.

Only the server-side needs to worry about security, the client is on the same host as the server so no special measures are needed to protect client-server communication. Security is needed for host-to-host traffic. To set up a secure static a client needs to supply a principal name and specify that it wants security enabled.

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 Heterogeneous Transports

Complete this section

Ensemble provides a flexible infrastructure for sending communication across a variety of different communication transports. Not only can different groups use different communication transports, but a single group can support communication on multiple transports at the same time.

The design of the transport module is split into three parts:

The socket module:

Low-level system calls: `send`, `sendto`, `recv` etc., implemented in a system-independent fashion. The `socket` directory contains the code. `socket/u` is a simple-minded implementation that uses the Ocaml Unix library directly. A more efficient version is located in `socket/s`, where native OS io-vector send/recv facilities are used.

Transports:

Self registering *transports*: Deering, UDP, TCP, NETSIM. These use the low-level socket module calls to provide an abstract *transport*.

Routers:

Uses a communication transport to build Ensemble specific send/recv capabilities. Length field, group id, and endpoint rank are added to each outgoing message. Basic parsing is performed on received messages and sender rank, group, and message length are extracted.

There are several *routers* in the `route` subdirectory. `signed.ml` adds a 16-byte MD5 checksum to each outgoing message. An agreed group-secret is used to key MD5, providing group authentication. Incoming messages are stripped of this header, and verified. `unsigned.ml` is the vanilla router.

The user can choose to use either one of the socket module implementations. The socket module interface is defined in `socket/socket.mli`. The unoptimized socket implementation (`usocket`) represents message data as a Caml string and benefits from native garbage collection. Its disadvantage is reduced performance. The optimized socket library (`ssocket`) uses native C io-vectors, and native operating-system scatter-gather message send/receive facilities. This provides much better performance, and zero-copy integration with C applications. The disadvantage is more difficult integration with native ML values.

The transports are defined in the `trans` subdirectory. UDP in `trans/udp.ml`, TCP in `trans/tcp.ml`, DEERING in `trans/ipmc`, and NETSIM in `trans/netsim`.

The `route` subdirectory contains three routes: `signed`, `unsigned`, and `bypass`.

7.1 Code walk-through

To provide better understanding of the design this section walks through a configuration of the unsigned router, UDP transport, and optimized socket library. We shall start from the bottom and work our way up.

In file `server/socket/s/nt/sendrecv.c`, there is code for sending an array of C io-vectors and part of an ML string for win32. The function takes five arguments:

- `info_v` : a structure describing a list of remote targets and a socket through which to send messages.
- `prefix_v` : an ML string that prefixes the data

- ofs_v, len_v: the offset and length of the prefix to send
- iova_a : an array of io-vectors wrapped in an ML representation

```

value skt_udp_mu_sendsv(
value info_v,
value prefix_v,
value ofs_v,
value len_v,
value iova_v
) {
    int naddr=0, i, ret=0, len=0;
    ocaml_skt_t sock=0 ;
    skt_sendto_info_t *info ;
    int nvecs = Wosize_val(iova_v) ;

    // Extract the set of addresses
    info = skt_Sendto_info_val(info_v);

    // Prepare the header
    skt_prepare_send_header(send_iova, peek_buf, Int_val(len_v), skt_iovl_len(iova_v));

    // Prepare the iovectors
    skt_add_ml_hdr(send_iova, 1, prefix_v, ofs_v, len_v);
    skt_gather(send_iova, 2, iova_v) ;

    sock = info->sock ;
    naddr = info->naddr ;

    for (i=0;i<naddr;i++)
// Send the message. Assume we don't block or get interrupted.
ret = WSASendTo(sock, send_iova, nvecs+2, &len, 0,
&info->sa[i], info->addrlen, 0, NULL);
if (SOCKET_ERROR == ret) skt_udp_error("skt_udp_mu_sendsv");

    return Val_unit;
}

```

The `_mu_` prefix is added to this function because it uses the ML/User convention for sending data. Each data packet is split into:

ML header length: Describes the length of the ML header. of length four bytes.

User data length: Describes the length of the user data. of length four bytes.

ML header: the ML header itself. Variable size.

User data: user data. Variable size.

The function builds a header of size eight that includes two integers: (a) ml-header length (b) io-vector length in network byte order. The header is the first in an array of io-vectors that includes in second place the ML-header, and then the array of user io-vectors. Once the io-array is assembled it is sent to each destination in the list using the native OS API.

`skt_udp_mu_sendsv` is hidden inside the socket library, and can safely be used using `Socket.udp_mu_sendsv`. The `sendto_info` structure can be created from an array of target socket addresses, and a sending socket.

```
type sendto_info
val sendto_info : socket -> Unix.sockaddr array -> sendto_info

val udp_mu_sendsv : sendto_info -> buf -> ofs -> len -> Iov.t array -> unit
```

The Hsys module makes access to `sendto`s safer, and changes its type:

```
val udp_mu_sendsv : sendto_info -> Buf.t -> ofs -> len -> Iovecl.t -> unit

(* Implementation *)
Socket.udp_mu_sendsv info
  (Buf.string_of buf) (Buf.int_of_len ofs) (Buf.int_of_len len)
  (Iovecl.to_iovec_array iovl)
```

Core Ensemble code, including the routers, does not use Socket calls directly. Rather, it uses the Hsys module which wraps all calls with a more type safe interface. Separate types are used for length, offset, io-vector, and buffer.

The UDP implementation at `trans/udp.ml` uses Hsys in the transmit function called `x`.

```
let x hdr ofs len iovl =
  Hsys.sendtosv dests hdr ofs len iovl;
  Iovecl.free iovl
```

The io-vector array is freed after the message is transmitted. The reference count for an iovec-array is decremented on two occasions: (1) it is sent on the network (2) it is handed to an application, and the callback has completed. The iovec refcount is initially set to one when the application sends it, and it is henceforth incremented whenever a copy of it created. Ultimately, the refcount will be decremented when the stability detection protocol determines that all group members received the message.

7.2 Design of the routers

Many endpoints belonging to different groups can coexist in a single Ensemble process. Each endpoint is identified by its connection identifier. The internal representation of this id is given in module `Conn`:

```

type id = {
  version      : Version.id ;
  group       : Group.id ;
  stack       : Stack_id.t ;
  proto       : Proto.id option ;
  view_id     : View.id option ;
  sndr_mbr    : sndr_mbr ;
  dest_mbr    : dest_mbr ;
  dest_endpt  : dest_endpt option
}

```

The id is mapped into a string using the `Route.pack_of_conn` function. Ensemble uses MD5 for this mapping. The probability of a collision, i.e., for two different endpoints to map onto a single string, is 2^{-64} which is sufficient for our purposes.

```

val pack_of_conn : Conn.id -> Buf.t

```

The purpose of the route module is to create a single interface to these various endpoints. The main type exported is `handlers`. This is essentially a large array holding the set of connection identifiers and the delivery function for each of them. When a message is received by the bottom-most part of the system, it is parsed by the socket code into an ML header that is a string, and the rest of the message which is received into a C-iovec. This information is later fed into the `deliver` function.

```

val deliver : handlers -> Buf.t -> Buf.ofs -> Buf.len -> Iovecl.t -> unit

```

Deliver takes the current set of handlers, and a message, figures out which endpoints need to receive this message and calls the appropriate handlers.

A transmission function is abstracted as a type `xmitf`:

```

(* transmit an Ensemble packet, this includes the ML part, and a
 * user-land iovecl.
 *)
type xmitf = Buf.t -> Buf.ofs -> Buf.len -> Iovecl.t -> unit

```

The Router module has an API allowing the creation of send/recv functions for connection-ids. It also allows installing and deleting such functions. The unsigned router is a simple example of using this functionality to create the basic, insecure, router. It defines function `f`:

```

val f : unit ->
  (Trans.rank -> Obj.t option -> Trans.seqno -> Iovecl.t -> unit) Route.t

```

This router will allow users to send (1) sender rank (2) ML object (3) sequence number and (4) a user iovec array. The body of the code calls `Route.create` where it mainly needs to define how it plans on handling `blast` and `merge`. Blast is how to send messages, merge is how to receive a message on behalf of several connection ids.

Part II

The Client

The client library (or simply the “client”) implements a message-passing protocol between server and user. The protocol used is described in the reference manual. The client-library has no internal threads. No message-memory is allocated by the client, all messages are allocated and freed by the user. This gives the user complete control on its memory foot-print. The client is thread-safe, several threads can send/rcv messages concurrently. Blocking socket operations are used to simplify client semantics.

In order to use the client-library the user application must first connect to the server. It can then create group members and perform a subset of Ensemble actions: Leave, Cast, Send, Send1, Suspect. There are other Ensemble operations that we decided not to support since they add more complexity than value.

The application must poll Ensemble periodically to see if there are any pending messages, and receive them. In the past, it was possible for the application not to receive messages while continuing to create new actions. This is now not possible. The application will be blocked at some point before flooding the server.

8 Java Application Interface

The API is constructed from a namespace named `Ensemble` and several public classes the major of which are: `View`, `JoinOps`, `Message`, `Connection`, and `Member`.

View: The `View` class describes a group membership view.

JoinOps: The `JoinOps` class contains a specification for a new member for `Ensemble` to create.

Message: The `Message` class describes a message received from `Ensemble`. A `Message` can be: a new `View`, a multicast message, a point-to-point message, a block notification, or an exit notification.

Connection: The `Connection` class implements the actual socket communication between the client and the server. It has three public methods the application has to use.

```
public class Connection {  
    public bool Poll();  
    public Message Recv();  
    public void Connect ();  
}
```

The application can open several `Ensemble` connections, however, a single connection should suffice. No action can be taken on a connection that has not connected to the server through the `Connect` call. Once connected the application can receive messages through the `Recv` method. `Recv` is a blocking call, in order to check first that there are pending messages the non-blocking `Poll` method should be used.

Member: The `Member` class embodies an `Ensemble` group member. A member can join a single group, no more. A `Member` can be in several states:

Pre: The initial status in which all members are created. The class constructor sets this as the default.

Joining: Joining a group is an asynchronous operation. A member is in the `Joining` state from the time it attempts to join, until when it receives a `View` message with the initial group membership.

Normal: Normal operation state. The member is a regular resident in the group. It can send/mcast messages and perform all other `Ensemble` operations.

Blocked: The member is currently blocked, and temporarily cannot perform any action. This state is achieved by sending a `BlockOk` in response to a `Block` request.

Leaving: The member has requested to leave the group. `Leave` is an asynchronous operation, the `Leaving` state captures the time between the `Leave` request and the final leaving of the group. It is possible for a member to receive a `Block` message after it has requested to leave the group. The member should not respond with a `BlockOk`, it is in the process of being removed from the group.

Left: The member has left the group and is in an invalid state

```

public class Member {
    public enum Status {
        Pre,          // the initial status
        Joining,      // we joining the group
        Normal,       // Normal operation state, can send/mcast messages
        Blocked,      // we are blocked
        Leaving,      // we are leaving
        Left          // We have left the group and are in an invalid state
    };

    public View current_view ;          // The current view
    public Status current_status = Status.Pre; // Our current status

```

The member state can be learned by examining the `current_status` field. The current view the member is part of is in the `current_view` field.

Prior to any action, the member has to join a group.

```

// Join a group with the specified options.
public void Join(JoinOps ops);

```

The set of operations allowed on a member in Normal state is:

Leave: Leave a group. This should be the last call made to the member. It is possible for messages to be delivered to this member after the call returns. However, it is illegal to initiate new actions on this member.

```

public void Leave();

```

Cast: Send a multicast message to the group.

```

public void Cast(byte[] data);

```

Send: Send a point-to-point message to a list of members.

```

public void Send(int[] dests, byte[] data);

```

Send1: Send a point-to-point message to the specified group member.

```

public void Send1(int dest, byte[] data);

```

Suspect: Report group members as failure-suspected.

```

public void Suspect(int[] suspects);

```

BlockOk: Send a BlockOk

```
public void BlockOk();
```

8.1 The client state-machine

Each group member moved inside a state-machine that has a very clear set of rules. Initially, it is in the **Pre** state. After asking to join a group, it is in the **Joining** state. When the first view arrives it is in the **Normal** state. In the **Normal** state and prior to a view-change Ensemble will send a **Block** message to the member. The member has to reply with a **BlockOk** action. After the **BlockOk** the member is in the **Blocked** state until the next view. When the next view is received it moves back to the **Normal** state. Upon a **Leave** request the member moves to the **Leaving** state which turns into **Left** after the **Exit** message arrives.

8.2 Locking

All **Connection** and **Member** method calls are thread-safe. However, accessing the public **Member** fields such as the **current_view** and **current_status** should be done while holding the connection lock. All Ensemble actions, except **Join**, can be performed only on a group-member that is in the **Normal** state. A multi-threaded application may need to synchronize its access to the client library so as to avoid a situation where one thread sends a **BlockOk** and another thread sends message on the group later. The connection lock should be used for synchronization purposes.

For example:

```
synchronization (conn)
{
    if (memb.current_status == Member.Status.Normal)
        memb.Cast("hello world");
    else
        System.out.Println("Blocked currently, please try again later");
}
```

Replying to **Block** message with a **BlockOk** and moving to the **Blocked** state can be done asynchronously. This gives the application a chance to send any pending messages prior to moving to the **Blocked** state. Depending on the application, this may allow the programmer to avoid locking.

8.3 The View structure

The view structure is composed of several fields describing the members of the current membership.

```

public class View {
    public int nmembers;
    public String version;    /** The Ensemble version */
    public String group;      /** group name */
    public String proto;      /** protocol stack in use */
    public int ltime;         /** logical time */
    public boolean primary;   /** this a primary view? */
    public String parameters; /** parameters used for this group */
    public String[] address;  /** list of communication addresses */
    public String[] endpts;   /** list of endpoints in this view */
    public String endpt;      /** local endpoint name */
    public String addr;        /** local address */
    public int rank;          /** local rank */
    public String name;        /** My name. */
    public ViewId view_id;    /** view identifier */
}

```

nmembers: The number of members.

version: The Ensemble version

group: The name of this group. This was chosen by the user when he joined the group.

proto: The names of all the layers in the Ensemble stack.

ltime: The logical time of the view.

primary: Is this a primary view?

parameters: The set of additional parameters used for this stack. These were chosen by the user when he joined the group.

address: The list of communication addresses of group members. Currently, this lists the addresses of *servers* not the clients.

endpts: list of endpoints in this view.

endpt: my endpoint name.

addr: my communication address.

rank: my rank.

name: My name. Does not change through the lifetime of this member.

view_id: The view identifier.

8.4 Join options

The join-options structure is used to specify which group an endpoint should join.

```

public class JoinOps {
    public String group_name = null;  /** group name.    */

    /** The default set of properties */
    public final String DEFAULT_PROPERTIES = "Vsync";
    /** requested list of properties. */
    public String properties = DEFAULT_PROPERTIES;

    public String parameters = null;  /** parameters to pass to Ensemble. */
    public String princ = null;       /** principal name */
    public boolean secure = false;    /** a secure stack? */
}

```

group_name: The group name to join.

properties: The set of properties requested. The most oftenly used property is Vsync which ensures virtually-synchronous point-to-point communication. The other commonly used properties are:

Total: totally ordered multicast messages

Auth: Authenticate members and MAC all group-messages.

Scale: use scalable protocols. Useful if there are more than 12 machines in the group.

See section 5.8 for more information.

parameters: An additional set of parameters controlling the stack.

princ: The client principal name. This is used only for secure stacks. It can be left null otherwise.

secure: Should the stack be secure?

8.5 Limitations

There are several limitations on argument sizes.

```

private final int ENS_DESTS_MAX_SIZE=      10;
private final int ENS_GROUP_NAME_MAX_SIZE= 64;
private final int ENS_PROPERTIES_MAX_SIZE= 128;
private final int ENS_PROTOCOL_MAX_SIZE=   256;
private final int ENS_PARAMS_MAX_SIZE=     256;
private final int ENS_ENDPT_MAX_SIZE=      48;
private final int ENS_ADDR_MAX_SIZE=       48;
private final int ENS_PRINCIPAL_MAX_SIZE=  32;
private final int ENS_NAME_MAX_SIZE=       ENS_ENDPT_MAX_SIZE+24;
private final int ENS_VERSION_MAX_SIZE=    8;
private final int ENS_MSG_MAX_SIZE=       32 *1024;

```

The important limitations are:

Number of destinations: The user can send a point-to-point message to a set of targets. The size of the target set is limited to `ENS_DESTS_MAX_SIZE`. The same goes for suspecting sets of members.

Message size: The maximal message size is limited to 32K.

8.6 Code examples

Take a look at the `Mtalk.java` program in the `client/java` subdirectory.

9 C Application Interface

The C-language API is similar in nature to the Java/C-sharp APIs. This section assumes the reader is already familiar with that material.

The main addition for C is the careful description of who allocates which memory.

In order for an application to start using Ensemble it needs to initialize a connection structure:

```
ens_conn_t *ens_Init(void);
```

The `ens_Init` call allocates and initializes an `ens_conn_t` structure that encapsulates a local socket connection to the Ensemble Server. All operations require the connection structure. Only one connection is needed for an application.

The application needs to poll the connection periodically to see if it has pending messages.

```
typedef enum ens_rc_t
    ENS_OK = 0,
    ENS_ERROR = 1
    ens_rc_t;

ens_rc_t ens_Poll(ens_conn_t *conn, int milliseconds, /*OUT*/ int *data_available);
```

`ens_Poll` returns an `ens_rc_t` return type which conveys whether the operations was successful or not. Poll takes a connection, and a number of milliseconds to wait for input on the socket. It is a blocking call, as is the rest of the API. If data is available the out argument: `data_available` will be set to 1; if no data is available it will be set to 0.

In order to Join a group the `ens_Join` call should be used.

```
ens_rc_t ens_Join(
    ens_conn_t *conn,
    ens_member_t *memb,
    ens_jops_t *ops,
    void *user_ctx
) ;
```

Join takes a connection, a member structure, a set of join-options and a opaque pointer for the user's use. It initializes the member structure, attaches the user-context to it, and sends a Join request to Ensemble. The set of allowed join-options is:

```
typedef struct ens_jops_t
    char group_name[ENS_GROUP_NAME_MAX_SIZE] ; /* The group name */
    char properties[ENS_PROPERTIES_MAX_SIZE] ; /* The set of properties */
    char params[ENS_PARAMS_MAX_SIZE] ; /* The set of parameters */
    char princ[ENS_PRINCIPAL_MAX_SIZE] ; /* My principal name (security) */
    int secure ; /* Do we want a secure stack (encryption + auth) */
    ens_jops_t ;
```

The user can choose the group-name, set of properties expected from the group, and a set of additional configuration parameters. The default values for properties is `ENS_DEFAULT_PROPERTIES`, the additional set of parameters should be empty for normal use. There are two parameters used

for security: the principal name, and the secure flag. If one is not interested in security, simply set the secure flag to zero. If the flag is set to one, then the principal name in PGP should be specified. The principal names for users are used for authentication purposes; only authenticated users are allowed into a secure group.

After joining a group there are several operations that are allowed on it: `Leave`, `Send1`, `Send`, `Cast`, `Suspect`, and `BlockOk`. All these calls are (a) blocking, they return only after the whole data has been written to the socket (b) thread-safe: they may be used from any application thread.

To leave a group use the `ens_Leave` call.

```
ens_rc_t ens_Leave(  
    ens_member_t *memb  
);
```

After the call the member structure becomes invalid and cannot be used for any other operations.

Point-to-point and multicast messages can be send with three different calls:

```
ens_rc_t ens_Cast(  
    ens_member_t *memb,  
    int len,  
    char *buf  
);  
  
ens_rc_t ens_Send(  
    ens_member_t *memb,  
    int num_dests,  
    int *dests,  
    int len,  
    char* buf  
);  
  
ens_rc_t ens_Send1(  
    ens_member_t *memb,  
    int dest,  
    int len,  
    char* buf  
);
```

For the above three calls The data is not freed nor allocated by the Ensemble client. The user needs to manager memory that is sent. Maximal message size is 32K. The maximal number of destinations is 10.

Report specified group members as failure-suspected. The maximal number of suspects is 10.

```
ens_rc_t ens_Suspect(  
    ens_member_t *memb,  
    int num,  
    int *suspects  
);
```


Tell the system we will no longer send messages in this view. Should be sent as a response to a Block message.

```
ens_rc_t ens_BlockOk(
    ens_member_t *memb
) ;
```

Messages can be received by the `ens_RecvMetaData` call together with `ens_RecvView` and `ens_RecvMsg`.

When there is data on the connection the `ens_RecvMetaData` tells which type of message has arrived and how much memory needs to be allocated for receiving it.

```
typedef enum ens_up_msg_t
    VIEW = 1,      /* A new view has arrived from the server. */
    CAST = 2,      /* A multicast message */
    SEND = 3,      /* A point-to-point message */
    BLOCK = 4,     /* A block request, prior to the installation of a new view */
    EXIT = 5       /* A final notification that the member is no longer valid */
ens_up_msg_t;

typedef struct ens_msg_t
    ens_member_t *memb;      /* endpoint this message belongs to */
    ens_up_msg_t mtype ;     /* message type */
    union
    struct
        /* The variant for VIEW: */
        int    nmembers; /* the number of members in a view */
        view;
    struct
        /* The variant for a point-to-point message */
        int    msg_size; /* length of a bulk-data */
        send;
    struct
        /* The variant for multicast message */
        int    msg_size; /* length of a bulk-data */
        cast;
        u;
    ens_msg_t;

ens_rc_t ens_RecvMetaData(ens_conn_t *conn, ens_msg_t *msg);
```

`ens_RecvMetaData` is a blocking call, therefore, it needs to be executed only after the user knows there is pending data on the socket. The user needs to pre-allocate an `ens_msg_t` structure which is used to store meta-information about the in-coming message:

- which member is this message for?
- what type is the message? view, point-to-point message, multicast message, block, or exit.
- For each type, how much memory is required to receive it.

The next step is to call `ens_RecvView` for a view-message, and `ens_RecvMsg` for a bulk-data message.

```
ens_rc_t ens_RecvView(ens_conn_t *conn,  
                      ens_member_t *memb,  
                      /*OUT*/ ens_view_t *view);  
  
ens_rc_t ens_RecvMsg(ens_conn_t *conn,  
                    /*OUT*/ int *origin, char *buf);
```

Acknowledgments

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