Ensemble Reference Manual

Mark Hayden, Ohad Rodeh *

December 18, 2002

Abstract

Ensemble is a reliable group communication toolkit implemented in the Objective Caml programming language. The purposes of this implementation are:

- to provide a concise and clear "reference" implementation of the Ensemble architecture and protocols
- to abstract protocol layer implementations as far as possible from the runtime system
- to support the application of formal methods to real implementations of distributed communication protocols
- to provide a flexible platform for ease of experimentation

Throughout, we attempt to follow a design that supports a simple compilation of the protocols to C. Two intermediate stages have recently been taken in the direction 1) the construction of a *native* C interface for Ensemble (CE) and 2) an implementation in C of the core Ensemble system (available from www.northforknet.com).

*Thanks to Takako Hickey, Roy Friedman, Robbert van Renesse, Zhen Xiao, and Ohad Rodeh for descriptions of their contributions.

Contents

1 Introduction

This document is attempting to serve several goals. It is intended to be:

- A description/motivation of the Ensemble architecture.
- An aid for learning about the Ensemble protocol layers and their workings.
- An informal specification of individual protocol layers and common compositions of the layers.
- Documentation of possible alternate designs/implementations of the system.
- A depository of specification and verification information developed for protocol layers.
- A depository of descriptions of potential projects.
- Source for combined hypertext code/text documentation of Ensemble.

Documentation TODO list:

- mark all ML values with mlval{}
- add more detail in the event fields section
- add "common problems" section
  - **EInit** vs. **EView**
- layer programming tutorial

Part I
The Ensemble Architecture

2 Identifiers

Ensemble uses a variety of identifiers for a variety of different purposes. Here we summarize the important ones and describe what they are used for. Their type definitions can be found in the **type** directory in the file corresponding to the name. Look in **type/README** for an up to date listing of these files. Most of the different identifiers support a similar interface for a variety of operations. Several of the identifiers are opaque, in the sense that the interface hides the actual structure of the identifier. All identifiers have a **string_of_id** function defined which gives a human-readable description of their contents.

Changes from Horus

- None of the identifiers are defined as fixed length sequences of bytes. This is something that remains to be done.
- Horus EID's have been split into endpoint and group identifiers.
- We have removed addressing information entirely from endpoint and group identifiers.
- Entity identifiers have been eliminated. Connection, stack, and protocol identifiers have been added to support a variety of features new to Ensemble.

2.1 Endpoint Identifiers

Endpoint identifiers are unique names for communication endpoints. A single process can create any number of local endpoint identifiers, each of which is guaranteed to be unique (within some limits). A process can have multiple endpoints in a single group. An endpoint can be a member of multiple groups. However, the endpoints in a group must be distinct.

2.2 Group Identifiers

Group identifiers serve as unique names of communication groups. They do not contain addressing information. The exception to this rule is that groups communicating via Deering multicast choose a random multicast address by taking a hash of the group address. Processes can create any number of local group identifiers, each of which is guaranteed to be unique (within some limits).

2.3 View Identifiers

View identifiers are unique identifiers of group views. Whenever communication protocols proceed through a view change, the resulting view is given a new view identifier. These are made unique by pairing the coordinator of the group with a logical timestamp that is advanced whenever a view change happens. Although two partitions of a group may share the same time stamp, they will have different coordinators.

2.4 Connection Identifiers

Connection identifiers are used to route messages to the precise destinations. They specify the exact destination endpoint or group, the view identifier, the protocol stack to deliver to, the type of protocol being used to send the message, as well as several other bits of information. Typically, endpoint or group identifiers are used to send messages to the correct processes and connection identifiers are used to route messages to the exact destination of a message within a process. Messages usually contain a connection identifier as a "header" of the message but do not contain endpoint identifiers or group identifiers, except as subfields of a connection identifier.

2.5 Protocol Identifiers

There is a one-to-one relationship between the standard protocol stacks of Ensemble and protocol identifiers. Applications select the protocol to use by specifying the protocol id of that stack. Having identifiers for protocols is convenient because they can be passed around in messages and have equality comparisons made on them, whereas the actual protocol stacks cannot.

2.6 Mode identifiers

Each communication domain has a corresponding mode identifier used to specify that domain.

2.7 Stack Identifiers

Stack identifiers are used to distinguish between the various domains that a protocol stack may be receiving messages through. Each of the various kinds of "channels" that protocol stacks use has a separate identifier. Currently, these are:

Primary : This is the primary communication channel for a protocol stack. This is normally where most messages are received.

Bypass : Messages sent via the optimized bypass protocols use this id.

Gossip : Messages sent by group merge protocols use this id.

Unreliable : This stack id is reserved for unreliable stacks.

3 The Event Module

Events in Ensemble are used for *intra-stack* communication, as opposed to inter-stack communication, for which messages are used. Currently, the event module is the only Ensemble-specific module that all layers use. Events contain a well-known set of fields which all layers use for communicating between themselves using a common *event protocol*. Learning this protocol is one of the harder parts of understanding Ensemble. In this section we describe the operations supported for events; in section 4 we describe the meaning of the various event types and their fields.

We repeatedly refer the reader to the source code of the event module source files, both **type/event.mli** and **type/event.ml**. This is done to ensure that information in this documentation does not fall out of date due to small changes in the event module.

Note that a certain number of the operations invalidate events passed as arguments to the function. This means that no further operations are accessing on the event should be done after the function call. The purpose of this limitation is to allow multiple implementations of the event module with different memory allocation mechanisms. The default implementation of events is purely functional and these rules can be violated without causing problems. Other implementations of the event module require that events be manipulated according to these rules, and yet other implementations trace event modifications to check that the rules are not violated. What this means is that protocol designers do not need to be concerned with allocation and deallocation issues, except in the final stages of development.

Currently a reference counting scheme is used for handling message bodies, which form the bulk of memory used in Ensemble. Reference counting is done by-hand, and events that reference Io-vectors must be freed using the free function (see below). The rest of the event is allocated on the ML heap, and is therefore freed automatically by the ML garbage collector.

3.1 Fields

Events are ML records with fixed sets of fields. We refer to **type/event.mli** for their type definitions and fields.

3.1.1 Extension fields

Events have a special field called the extension field. Uncommon fields are included in up events as a linked list of extensions to this field. The list of valid extensions is defined in **type/event.mli** by the type definition **fields**.

3.1.2 Event Types

Events have a "type" field (called **typ** to avoid clashes with the **type** keyword) in them which can take values from a fixed set of enumerated constants. For the enumerations of the type fields for events, we refer to **appl/event.mli** for the type definitions for **typ**.

3.1.3 Field Specifiers

Events have defined for them a variant record called **field**. These are called field specifiers. There is a one-to-one relation between the fields in up and down events and the variants in the fields specifiers. As will be seen shortly, lists of field specifiers are passed to event constructor and modifier functions to specify the fields in an event to be modified and their values. This allows changes to an event to be specified incrementally.

3.2 Constructors

Events are constructed with the **create** function.

```
  (* Constructor *)
val create : debug -> typ -> field list -> t
```

Create takes 3 arguments:

- The string name of the module or location where this operation is being performed. This is used only for debugging purposes and usually the value **name** (defined to be the name of the current module) is used for this argument.
- The type of the event, which is a **typ** enumeration.
- A list of field specifiers for changing the values of the fields in the new events. Unmodified fields should not be accessed. For example, if an empty list is passed as a field specifier then only the type field of the event will be available in the event.

The return value of the constructor functions is a valid event.

3.3 Special Constructors

type/event.ml defines some special case constructors for either performance or ease-of-coding reasons. All of these constructors are defined using the **create** function or could be defined using them.

3.4 Modifiers

Events are modified with the **set** function.

```
  (* Modifier *)
val set : debug -> t -> field list -> t
```

set takes 3 arguments:

- The string name where this modification is taking place. Used only for debugging purposes.
- The event which is being modified. The event passed as an argument to this function is invalidated: no further references should be made to the event.
- A field specifier list. See the arguments description for Constructors.

The return value of **set** is a new event with the same fields as the original event, except for the changes in the specifier list.

3.5 Copiers

Events are copied with the **copy** function.

```
  (* Copier *)
val copy : debug -> t -> t
```

Copy takes two arguments:

- The name where this modification is taking place. Used only for debugging purposes.
- The event to be copied.

The return value is a new event with its fields set to the same values as the original.

3.6 Destructors

Events are released with the **free** function.

```
  (* Destructor *)
val free : debug -> t -> unit
```

Free functions takes two arguments:

- The name where this modification is taking place. Used only for debugging purposes.
- The event to be deallocated. This event becomes invalidated by this function call. No further references to the event should be made.

The return value is the **unit** value.

4 Event protocol: Intra-stack communication

Ensemble embodies two forms of communication. The first is communication between protocol stacks in a group, using messages sent via some communication transport. The second is intra-stack communication between protocol layers sharing a protocol stack (see fig 1), using Ensemble events (see page 8 for a overview of Ensembleevents). One use of events is for passing information directly related to messages (i.e., broadcast messages are usually associated with **ECast** events). However, events also are used for notifying layers of group membership changes, telling other layers about suspected failed members, synchronizing protocol layers for view changes, passing acknowledgment and stability information, alarm requests and timeouts, etc.... In order for a set of protocol layers to harmoniously implement a higher level protocol, they have to agree on what these various events "mean," and in general follow what is called here the Ensemble *event protocol*.

The layering in Ensemble is advantageous because it allows complex protocols to be decomposed into a set of smaller, more understandable protocols. However, layering also introduces the event protocol which complicates the system through the addition of intra-stack communication (the event protocol) to inter-stack communication (normal message communication).

Be aware that this information may become out of date. Although the "spirit" of the information presented here is unlikely to change in drastic ways, always consider the possibility that this information does not exactly match that in **type/event.ml** *and* **type/event.mli**. *Please alert us when such inconsistencies are discovered so they may be corrected.*

The documentation of the event protocol proceeds as follows.

- "types" of events are listed along with a summary of their meaning
- [**TODO: the types that usually have a message associated with them are identified**]
- event fields are described along with a summary of their usage
- [**TODO: a table is given showing the event types for which the various event fields have defined values**]
- the several *event chains* which occur in protocol stacks are listed (event chains are sequences of event micro-protocols that tend to occur in Ensemble protocol stacks)

4.1 Event Types

This section describes the different types of events. See fig 2 for the source code of enumerated types. The behavior of a layer depends not only on the event type and its fields, but also on the *direction* from which it arrives. For example, an **ESend** event travels in the sender stack from the application down, and at the receiver from the bottom, up to the application. The sender and receiver layers behave quite differently depending on whether the message is sent or received. In what follows, we sometimes specifically include the event direction. Detailed Descriptions:

- **Up(EBlock)**: The coordinator is blocking the view. Is a reply to **Dn(E()**Block); replied with **Dn(EBlockOk)**.
- **Dn(EBlock)**: The group is being blocked. Is a reply to **Up(ESuspect)** and **Up(EMergeRequest)**; replied with **Up(EBlock)**.
- **Up(EBlockOk)**: The coordinator gets one of these events when the group is blocked. Is a reply to **Dn(EBlockOk)**; replied with **Dn(EView)** or **Dn(EMergeRequest)**.

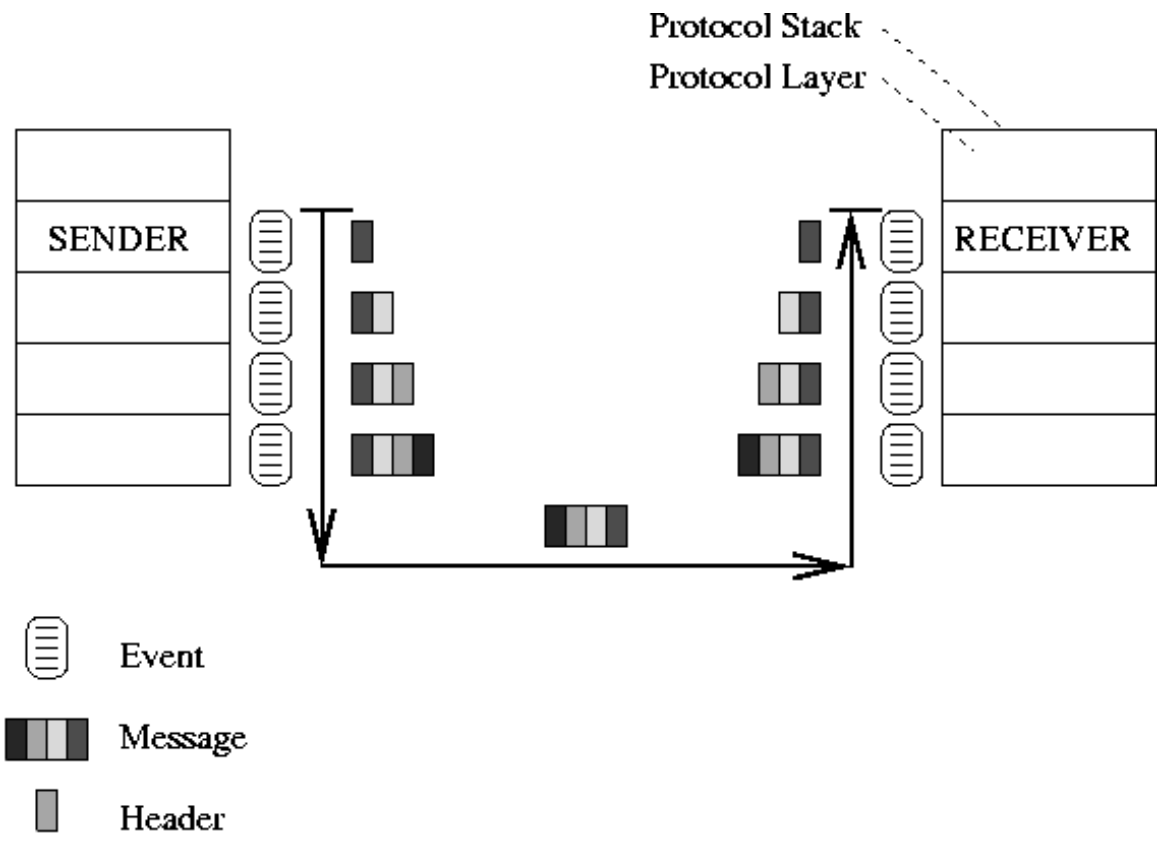

Figure 1: *Events are used for intra-stack communication: layers can only communicate with other layers by modifying events; layers never read or modify other layer's message headers. Messages are used for inter-stack communication: only messages are sent between group members; events are never sent between members.*

- **Dn(EBlockOk)**: Is a reply to **Up(EBlock)**; replied with **Up(EBlockOk)** (but usually only at the coordinator).
- **Up(ECast)**: A member (whose rank is specified by the origin field) broadcast a message to all members in the group. Usually the broadcast is delivered at all members except the sender.
- **Dn(ECast)**: A message is being broadcast. Replied with **Up(ECast)** at all members but sender.
- **Up(ESend)**: Another member sent us a pt2pt message.
- **Dn(ESend)**: A message is being sent pt2pt. Results in an **Up(ESend)** at the destination.
- **ESubCast**: A message that will be multicast to a subset of the group.
- **ECastUnrel**: A message that will be unreliablely multicasted
- **ESendUnrel**: A message that will be unreliablely sent point-to-point

```
    (* These events should have messages associated with them. *)
  | ECast (* Multicast message *)
  | ESend (* Pt2pt message *)
  | ESubCast (* Multi-destination message *)
  | ECastUnrel (* Unreliable multicast message *)
  | ESendUnrel (* Unreliable pt2pt message *)
  | EMergeRequest (* Request a merge *)
  | EMergeGranted (* Grant a merge request *)
  | EOrphan (* Message was orphaned *)

    (* These types do not have messages. *)
  | EAccount (* Output accounting information *)
(*| EAck       *)(* Acknowledge message *)
  | EAsync (* Asynchronous application event *)
  | EBlock (* Block the group *)
  | EBlockOk (* Acknowledge blocking of group *)
  | EDump (* Dump your state (debugging) *)
  | EElect (* I am now the coordinator *)
  | EExit (* Disable this stack *)
  | EFail (* Fail some members *)
  | EGossipExt (* Gossip message *)
  | EGossipExtDir (* Gossip message directed at particular address *)
  | EInit (* First event delivered *)
  | ELeave (* A member wants to leave *)
  | ELostMessage (* Member doesn't have a message *)
  | EMergeDenied (* Deny a merge request *)
  | EMergeFailed (* Merge request failed *)
  | EMigrate (* Change my location *)
  | EPresent                          (* Members present in this view *)
  | EPrompt (* Prompt a new view *)
  | EProtocol (* Request a protocol switch *)
  | ERekey (* Request a rekeying of the group *)
  | ERekeyPrcl (* The rekey protocol events *)
  | ERekeyPrcl2 (*                          *)
  | EStable (* Deliver stability down *)
  | EStableReq (* Request for stability information *)
  | ESuspect (* Member is suspected to be faulty *)
  | ESystemError (* Something serious has happened *)
  | ETimer (* Request a timer *)
  | EView (* Notify that a new view is ready *)
  | EXferDone (* Notify that a state transfer is complete *)
  | ESyncInfo
      (* Ohad, additions *)
  | ESecureMsg (* Private Secure messaging *)
  | EChannelList (* passing a list of secure-channels *)
  | EFlowBlock (* Blocking/unblocking the application for flow control*)
(* Signature/Verification with PGP *)
  | EAuth
```



```
  | ESecChannelList                    (* The channel list held by the SECCHAN layer *)
  | ERekeyCleanup
  | ERekeyCommit
```

- **Up(EMergeRequest)**: Some other partition want to merge with us. Replied with **Dn(EMergeGranted)** or **Dn(EMergeDenied)**.
- **Up(EMergeGranted)**: Notification that a merge is ready to proceed.
- **Dn(EMergeGranted)**: Done by the coordinator after an **Up(EMergeRequest)** to tell the other coordinator that the merge is progressing. Results in an **Up(EMergeGranted)** at the merging coordinator.
- **Up(EMergeDenied)**: This is notification that the coordinator of a partition we tried to merge with has explicitly denied the merge. Is a reply to **Dn(EMergeRequest)**; replied with **Dn(E()**View).
- **Dn(EMergeDenied)**: Done by the coordinator after an **Up(EMergeRequest)** to tell another coordinator that its request has been denied. Is a reply to **Up(EMergeRequest)**.
- **Up(EMergeFailed)**: This is notification that some problem occurred in an attempt to merge with another partition of our group. Is a reply to **Dn(EMergeRequest)**; replied with **Dn(E()**View).
- **Up(EOrphan)**: A message has lost its parent. Usually it is OK to ignore this message: it is just being delivered in case we are interested.
- **EAccount**: Used to control accounting information. Periodically this event is passed through the stack, and each layer can record its current information/performance.
- **EAsync**: Used to handle asynchronous events. Unused currently.
- **Up(EDump)**: A layer wants the stack to dump its state. Usually the top-most layer will bounce this down as a **Dn(EDump)** and the members will dump their state on the **Dn(EDump)** event.
- **Dn(EDump)**: Dump your state. Pass it on. Is a reply to **Up(EDump)**.
- **Up(EElect)**: This member has been elected to be coordinator of the group. Usually means that this member will be generating failure and view events and managing the group for the rest of the view.
- **Up(EExit)**: This protocol stack has been disabled (because of a previous **Dn(ELeave)** event). Layers should pass this event up and then do nothing else.
- **Up(EFail)**: This is notification that some members have been marked as failed. This does not necessarily mean we will get no more messages from the failed members. The COM layer drops them, but messages from those members retransmitted by other members are still delivered. Usually, it also means the coordinator has started or will start a new view soon (but this is not necessary). Is a reply to **Dn(EFail)**.
- **Dn(EFail)**: Some members are being failed. Is a reply to **Up(ESuspect)**; replied with **Up(EFail)**.
- **Up(EGossipExt)**: A gossip message has arrived. Note that the data is in the extension fields.

- **Dn(EGossipExt)**: Transmit a gossip message. Note that data is carried in extension fields of events and does not use the normal header pushing mechanism.
- **Up(EGossipExtDir)**: The same, but send a gossip message to a specified address.
- **Up(EInit)**: This is the first event that any layer receives. It should be passed up the stack.
- **Up(ELeave)**: Some member (specified by the origin field) is leaving the group.
- **Dn(ELeave)**: We are leaving the group. Replied with **Up(EExit)** event. Depending on when this is seen it can mean different things. Before a new view change it means we are really leaving the group. After a new view, it usually means that we have a new protocol stack taking part in the next view and the **Dn(ELeave)** is just garbage collecting this protocol stack because its view is over.
- **Up(ELostMessage)**: This is notification that some protocol layer below does not have a message it thinks it should have. What this means is usually highly-protocol-stack-specific. Sometimes replied with **Dn(EFail)**.
- **EMigrate**: Used to support migration of an endpoint between transport addresses.
- **EPresent**: A notification that all group members currently in the group are live and started "talking". This is useful in large groups, where it takes time for *all* members to synchronize and agree on the view.
- **EPrompt**: Ask for a new view
- **EProtocol**: Ask for a new protocol
- **ERekey**: Request a rekey
- **ERekeyPrcl**: Used for inter-layer communication between the rekeying layers. Used to separate out their event types.
- ERekeyPrcl2: dito.
- **Up(EStable)**: This event contains stability information. If we are buffering broadcast messages, we can use this to decide which messages are safe to drop.
- **EStableReq**: Ask for stability information
- **Up(ESuspect)**: This is notification that some other layer (or possibly some other member) thinks that some members should be kicked out of the group. Replied with **Dn(EFail)** and often **Dn(EBlock)**.
- **Dn(ESuspect)**: Some members are suspected to be failed. Replied with an **Up(ESuspect)** with the same members failed.
- **Up(ESystemError)**: Something serious has happened. Do whatever you feel like because the world is about to fall apart.
- **Up(ETimer)**: A timer has expired. Pass it on.

- **Dn(ETimer)**: A request for a timer alarm. Replied with an **Up(ETimer)** at or after the requested time.
- **Up(EView)**: A new view is ready. Note that this does not affect our protocol: usually a different instance of our protocol stack will be created to take care of the next view. This event is not delivered at the beginning of a view. The **Up(EView)** event signals the end of a view. **Up(EInit)** events are delivered at the beginning of a view.
- **Dn(EView)**: A new view is prepared. Usually followed by an **Up(EView)**. Usually does not affect the current protocol stack, but later results in the creation of a new protocol stack for the new view.
- **EXferDone**: Used for the state-transfer layer. Tells the Xfer-layer that this endpoint has completed its state-transfer protocol.
- **ESyncInfo**: Used inside the virtual-synchrony protocol.
- **ESecureMsg**: Send a secure private message on an encrypted point-to-point channel to another member.
- **EChannelList**: Used to debug the secure-channel layer. The event lists the set of current open secure-channels.
- **ESecChannelList**: dito.
- **EFlowBlock**: Blocking/unblocking the application for flow control.
- **EAuth**: Signature/Verification with PGP.
- **ERekeyCleanup**: Erase all current open secure point-to-point connections. This initializes the secure-channel cache.
- **ERekeyCommit**: Commit the current tentative group-key as the key for the upcoming view.

4.2 Event fields

Here we describe all the fields that appear in the events. The type definitions appear in fig **??** and fig 3. Default values for the fields appear in fig **??**.

4.2.1 Event Fields

- Typ: The type of the event.
- Flags: A bitfield specifying a set of potential flags for the event.
- Peer: rank of sender/destination
- Iov: Iovec list containing raw application data.
- ApplMsg: was this message generated by an appl? This sometimes requires a different treatment than other, system generated messages.

```
type field =
      (* Common fields *)
  | Type        of typ            (* type of the message*)
  | Peer        of rank           (* rank of sender/destination *)
  | Iov         of Iovecl.t       (* payload of message *)
  | ApplMsg                       (* was this message generated by an appl? *)

      (* Uncommon fields *)
  | Address     of Addr.set   (* new address for a member *)
  | Failures    of bool Arrayf.t  (* failed members *)
  | Presence    of bool Arrayf.t  (* members present in the current view *)
  | Suspects    of bool Arrayf.t  (* suspected members *)
  | SuspectReason of string   (* reasons for suspicion *)
  | Stability   of seqno Arrayf.t (* stability vector *)
  | NumCasts    of seqno Arrayf.t (* number of casts seen *)
  | Contact     of Endpt.full * View.id option (* contact for a merge *)

      (* HEAL gossip *)
  | HealGos     of Proto.id * View.id * Endpt.full * View.t * Hsys.inet list
  | SwitchGos   of Proto.id * View.id * Time.t  (* SWITCH gossip *)
  | ExchangeGos of string (* EXCHANGE gossip *)

      (* INTER gossip *)
  | MergeGos    of (Endpt.full * View.id option) * seqno * typ * View.state
  | ViewState   of View.state (* state of next view *)
  | ProtoId     of Proto.id (* protocol id (only for down events) *)
  | Time        of Time.t (* current time *)
  | Alarm       of Time.t (* for alarm requests *)
  | ApplCasts   of seqno Arrayf.t
  | ApplSends   of seqno Arrayf.t
  | DbgName     of string

      (* Flags *)
  | NoTotal                    (* message is not totally ordered*)
  | ServerOnly                (* deliver only at servers *)
  | ClientOnly                (* deliver only at clients *)
  | NoVsync
  | ForceVsync
  | Fragment                (* Iovec has been fragmented *)

      (* Debugging *)
  | History     of string       (* debugging history *)

      (* Private Secure Messaging *)
  | SecureMsg of Buf.t
  | ChannelList of (rank * Security.key) list

      (* interaction between Mflow, Pt2ptw, Pt2ptwp and the application *)
  | FlowBlock of rank option * bool

      (* Signature/Verification with Auth *)
  | AuthData of Addr.set * Auth.data
```

- Address: A address for a member, used, for example, in sending gossip messages to specific endpoints.
- Failures: List of ranks of members that have failed.
- Presence: The list of members present in the current view. Used in large groups.
- Suspects: List of ranks of members that are suspected to have failed or be faulty in some way.
- SuspectReason: String containing "reason" for suspecting members. Used for debugging purposes.
- Stability: Vector of number of broadcasts for each member in the group that are stable.
- NumCasts: Vector of number of known broadcasts for members in the group.
- Contact: Endpoint of contact used for communication to endpoints outside of group. Usually only in merge events.
- HealGos: The type of gossip messages sent between HEAL layers.
- SwitchGos: dito, for SWITCH layers.
- ExchangeGos: dito, for EXCHANGE layers.
- MergeGos: dito, for MERGE layers.
- ViewState: Used to pass around view state for new views
- ProtoId: A new protocol id that we are switching to.
- Time: Time that the event/message was received.
- Alarm: Requested time for an alarm.
- ApplCasts: The sequence numbers of the latest multicast messages sent in the group.
- ApplSends: dito, for send messages
- NoTotal: Flag: specifying that this message should not be totally ordered even if total a totally-ordered stack is in use. Used to send Ensemble control messages which should not be delayed. Fail and View messages are sent this way.
- ServerOnly: Flag: Deliver only at servers.
- ClientOnly: Flag: Deliver only at clients.
- NoVsync: Flag: This message is not subject to virtual synchrony
- ForceVsync: Flag: this message *must* be subject to virtual synchrony
- Fragment: Flag: This event contains a fragment of an Iovec.
- SecureMsg: Send this buffer through a secure-channel point-to-point to another member.
- ChannelList: Describe the secure channel list.

- FlowBlock: Used for interaction the flow control layers: Mflow, Pt2ptw, Pt2ptwp and the application
- AuthData: Send a buffer for PGP to check, and receive a checked buffer from PGP

4.3 Event fields and the "types" for which they are defined

[TODO]

4.4 Event Chains

We describe here common event sequences, or chains, in Ensemble. Event chains are sequences of alternate up and down events that ping-pong up and down the protocol stack bouncing between the two end-layers of the chain. The end layers are typically the the top and bottom-most layers in the stack (eg., TOP and BOT). The most common exceptions to this are the message chains (Sends and Broadcasts), which can have any layer for their top layer.

Note that these sequences are just prototypical. Necessarily, there are variations in which of layers see which parts of these sequences. For example, consider the Failure Chain in a virtual synchrony stack with the GMP layer. The Failure Chain begins at the coordinator with an **ESuspect** event initiated at any layer in the stack. The BOT layer bounces this up as an **ESuspect** event. The top-most layer usually responds with a **EFail** event. The **EFail** event passes down through all the layers until it gets to the GMP layer. The GMP layer at the coordinator both passes the **EFail** event to the layer below and passes down a **ECast** event (thereby beginning a Broadcast Chain...). At the coordinator, the **EFail** event bounces off of the BOT layer as an **EFail** event and then passes up to the top of the protocol stack. At the other members, an **ECast** event will be received at the GMP layer. The message is marked as a "Fail" message, so the GMP layers generate and send down an **EFail** event (just like the one at the coordinator) and this is also bounced off the BOT layer as an **EFail** event. The lesson here is that the different layers in the different members of the group all essentially saw the same Failure Chain, but exact sequencing was different. For example, the layers above the GMP layer at the members other than the coordinator did not see a **EFail** event. **[TODO: give diagram]**

[TODO: Leave Chain]

4.4.1 Timer Chain

Request for a timer, followed by an alarm timeout.

ETimer	down: timeout requested, sent down to BOT.
ETimer	up: alarm generated in BOT at or after requested time, and sent up.

4.4.2 Send Chain

Send a pt2pt message followed by stability detection.

ESend	down: send a pt2pt message down.
ESend	up: destinations receive the message
EStable	message eventually becomes stable, and stability information is bounced off BOT.

4.4.3 Broadcast Chain

Broadcast of a message followed by stability detection.

ECast	down: broadcast a message
ECast	up: other members receive the broadcast
EStable	broadcast eventually becomes stable, and stability information is bounced off BOT

4.4.4 Failure Chain

Suspicion and "failure" of group members.

ESuspect	down: suspicion of failures generated at any layer
ESuspect	up: notification of suspicion of failures
EFail	down: coord fails suspects
EFail	up: all members get failure notice

4.4.5 Block Chain

Blocking of a group prior to a membership change.

ESuspect/EMergeRequest	up: reasons for coord blocking
EBlock	down: coord starts blocking
EBlock	up: all members get block notice
EBlockOk	down: all members reply to block notice
EBlockOk	up: coord get block OK notice
EMergeRequest EView	down: coord begins Merge or View chain

4.4.6 View Chain

Installation of a new view, followed by garbage collection of the old view.

EView	down: coord begins view chain (after failed merge or blocking)
EView	up: all members get view notice
EExit	down: protocol stacks are ready for garbage collection [**todo**]
EExit	up: protocol stacks are garbage collected

4.4.7 Merge Chain (successful)

Partition A merges with partition B, followed by garbage collection of the old view. We focus on partition A and only give a subset of events in partition B.

EMergeRequest	down: coord A begins merge chain (after blocking)
EMergeRequest	up: coord B gets merge request
EMergeGranted	down: coord B replies to merge request
EMergeGranted	up: coord A gets merge OK notice
EView	down: coord A installs new view for coord B
EView	up: all members in group A get view notice
EExit	down: protocol stacks are ready for garbage collection
EExit	up: protocol stacks are garbage collected

[**TODO: EExit above is currently ELeave**]

4.4.8 Merge Chain (failed)

Failed merge, followed by installation of a view.

EMergeRequest	down: coord begins merge chain (after blocking)
EMergeFailed or	
EMergeDenied	up: coord detect merge problem
EView	down: coord begins view chain

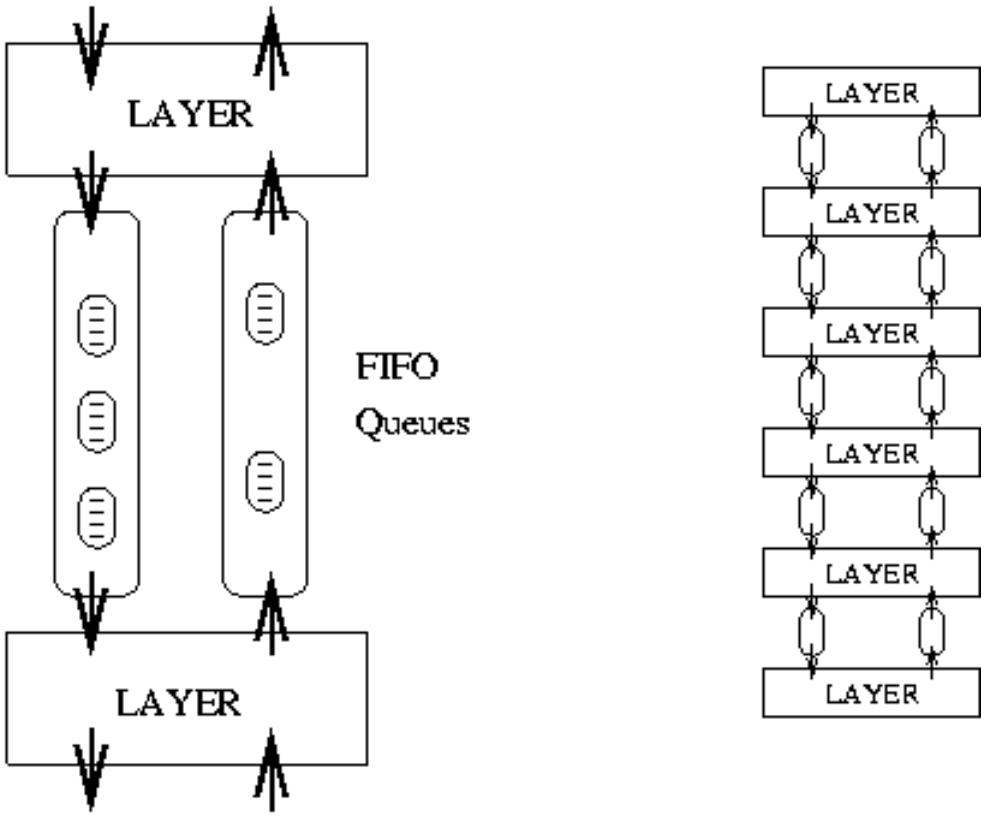

Figure 4: *Layers are executed as I/O automata, with pairs FIFO event queues connecting adjacent layers.*

5 Layer Execution Model

5.1 Callbacks

Layers are implemented as a set of callbacks that handle events passed to the layer by Ensemble from some other protocol layer. These callbacks can in turn call callbacks that Ensemble has provided the layer for passing events onto other layers. Logically, a layer is initialized with one callback for passing events to the layer above it, and another callback for passing events to the layer below it. After initialization, a layer returns two callbacks to the system: one for handling events from the layer below it and another for handling events from the layer above it. In practice, these "logical callbacks" are subdivided into several callbacks that handle the cases where different kinds of messages are attached to events.

5.2 Ordering Properties

The system infrastructure that handles scheduling of protocol layers and the passing of events between protocol layers provides the following guarantees:

FIFO ordering : The infrastructure guarantees that events passed between two layers are delivered in order. For instance, if layer A is stacked above layer B, then all events layer A passes to layer B are guaranteed to be delivered in FIFO order to layer B. In addition, events that

layer B passes up to layer A are guaranteed to be delivered in FIFO order to layer A. Note that these ordering properties allow the scheduler some flexibility in scheduling because they only specify the ordering of events in a single channel between a pair of layers.

no concurrency : The sytem infrastructure that hands events to layers through the callbacks never invokes a layer's callbacks concurrently. It guarantees that at most one invocation of any callback is executing at a time and that the current callback returns before another callback is made to the protocol layer. See fig 4 for a diagram of layer automata. Note that although a single layer may not be executed concurrently, different layers *may* be executed concurrently by a scheduler.

The execution of a protocol stack can be visualized as a set of protocol layers executing with a pair of event queues between each pair: one queue for events going up and another for events going down. The protocol layers are then automata that repeatedly are scheduled to take pending events from one of the adjacent incoming queues, execute it, and then deposit zero or more events into two adjacent outgoing queues (see fig 4).

6 Layer Anatomy: what are the pieces of a layer?

This is a description of the standard pieces of a Ensemble layer. This description is meant to serve as a general introduction the standard "idioms" that appear in layers. Because all layers follow the same general structure, we present a single documentation of that structure, so that comments in a layer describe what is particular to that layer rather than repeating the features each has in common with all the others. Comments on additional information that would be useful here would be appreciated.

6.1 Design Goals

A design goal of the protocol layers is to include as little Ensemble-specific infrastructure is present in the layers. For instance, none of the layers embody notions of synchronization, messages operations, of event scheduling. In fact, the only Ensemble-specific modules used by layers are the Event and the View modules.

6.2 Notes

Some general notes on layers:

- All layers are in single files.
- Usually the only objects exported by a layer are the type "header" and the value "l". When referred to outside of a layer, these values are prefixed with the name of the layer followed by "." and the name of the object (either "header" or "l"). For instance the STABLE layer is referenced by the name "Stable.l".

6.3 Values and Types

Listed below are the values and types commonly found in a layer, listed in the usual order of occurrence. For each object we give a description of its typical use and whether or not it is exported out of the layer.

- **name** : Local variable containing the name of the layer.
- **failwith** : Typically this standard Objective Caml function is redefined to prefix the failure message with the name of the layer. Sometimes it is also redefined to dump the state of a layer.
- **header** : Exported type of the header a layer puts on messages. Layers that do not put headers on messages do not have this type defined. The type is exported abstractly so it is opaque outside the layer. This type is usually a disjoint union (variant record) of several different types. Some example variants are:
  - **NoHdr** : Almost always one of the values is defined to be **NoHdr** which is used for messages for which the layer does not put on a (non-trivial) header
  - **Data** : Put on application data messages from the layer above
  - **Gossip** : Put on gossip messages (such as by a stability layer)
  - **Ack/Nak** : Acknowledgement or negative acknowledgements

- **Retrans** : Retransmissions of application data messages
- **View** : List of members in a new view.
- **Fail** : List of members being failed.
- **Suspect** : List of members to be suspected.
- **Block** : Prepare to synchronize this group.

- **nohdr** : exported variable. This is a variable that only occurs in a few layers. It is always defined to be the value **NoHdr** of the header type of the layer. It is exported so that the rest of the system can do optimizations when layers put trivial headers on a message.

- **normCastHdr** : exported variable. This is a variable that is used only by special layers that may need to generate a valid header for this layer. This variable should have no relevence to the execution of a layer.

- **state** : Local type that contains the state of an instance of a layer. Some layers do not yet use this, but eventually all of them will. [**layers that do not use a state varable have the state split up amongst several local state variables**]. The state then is referenced through the local variable **s**. A field in a state record is referred to by the Objective Caml syntax, **s.field**, where **field** is the name of a field in the state record.

  Some example field names used in layer states:

  - **time** : time of last (up) **ETimer** event seen
  - **next_sweep/next_gossip** : next time that I want to do something (such as retransmit messages, synchronize clocks, ...)
  - **sweep** : time between sweeping
  - **buf** : buffer of some sort
  - **blocking/blocked** : boolean for whether group is currently blocking
  - **ltime** : the current "logical time stamp" (usually taken from the **view_id**)
  - **max_ltime** : the largest logical time stamp seen by this member
  - **seqno** : some sequence number
  - **failed** : information on which members have failed (either a list of ranks or a boolean vector indexed by rank)
  - **elected** : do I think I am the coordinator?

  General notes on fields:

  - *fields with type vect*: usually array with one entry for each member, indexed by rank
  - *fields with type map*: usually mapping of members eid to some state on the member
  - *fields with "up" (or "dn") in the name*: refers to some info kept on events that are going up (or down)
  - *fields with "cast" (or "send") in the name*: refers to state kept about broadcasts (or sends)
  - *fields with "buf" in the name*: refers to some buffer
  - *fields with "dbg" in the name*: fields used only for debugging puposes

  - *fields with "acct" in the name*: fields used only for keeping track of tracing or accounting data for the layer

- **dump** : Local function that takes a value of type **state** and prints some representation for debugging purposes

- **member** : Local type that only occurs in some layers. Defines state kept for each member in a group. Typically, the layer's state will have an array of member objects indexed by rank (and this field is usually called **members**). The notes above fields in state records generally apply to the fields in member types as well.

- **init** : Initialization function for this layer. Takes two arguments. The first is a tuple containing arguments specific to this layer. The second is a view state record containing arguments general to the protocol stack this layer is in. This function does any initialization required and returns a **state** record for a layer instance.

  Some example names of initialization parameters for layers:

  - **sweep** : float value of how often to carry out some action (such as retransmitting messages or pinging other members)
  - **timeout** : amount of time to use for some timeout (such as how long to wait before kicking a non-responsive member out of a group)
  - **window** : size of window to use for some buffer

- **hdlrs** : Function initializing handlers for this layer. Takes two arguments. The first is a **state** record for this layer. The second is a record containing all the handlers the layer is to use for passing events out of the layer. Return value is a set of handlers to be used for passing events into this layer.

name	in/ out	up/ dn	above/ below	message?	header?
upnm	out	up	above	no	no
up	out	up	above	yes	no
dnnm	out	dn	below	no	no
dnlm	out	dn	below	no	yes
dn	out	dn	below	yes	yes
upnm	in	up	below	no	no
uplm	in	up	below	no	yes
up	in	up	below	yes	yes
dnnm	in	dn	above	no	no
dn	in	dn	above	yes	no

Table 1: *The 10 standard event handlers.*

7 Event Handlers: Standard

Logically, a protocol has two incoming event handlers (one each above and below) and two outgoing event handlers (one each above and below). In practice, because some events have messages and others do not, these handlers are split up into several extra handlers. The breakdown of the 4 logical handlers into 10 actual handlers is done for compatibility with the ML typechecker. Typechecking is used extensively to guarantee that layers recieve messages of the same type they send. This is a very useful property because it prevents a large class of programming errors.

In the standard configuration, each layer has 10 handlers. A handler is uniquely specified by a set of characteristics: whether it is an incoming or outgoing handler, a handler for up events or down events, a handler for communication with the layer above or for the layer below, whether it has an associated message, and whether it has an associated header. See table 1 for a enumeration of the 10 handlers. Of the 10 handlers, 5 are outgoing and 5 are incoming; 5 are up event handlers and 5 are down event handlers; 4 are for event communication with the layer below and 6 are for event communication with the layer above. These are depicted in fig 5.

The names of the handlers have two parts. The first specifies the sort of event the handler is called with ("up" or "dn"). The second specifies the sort of message that is associated with the event and may be either "" (nothing, the default case), "lm" (for local message), or "nm" (for no message), which correspond to:

nothing: Events with associated messages, where the message was created by a layer above this layer. This layer was not the first layer to push a header onto the message and will not be the last layer to pop its header off the message.

"lm": Events with associated messeges, where the message was created by this layer. This was the first layer to push a header onto the message and is the last layer to pop its header off of the message.

"nm": Corresponds to events without associated messages. These handlers always take a single argument which is either an up event or a down event.

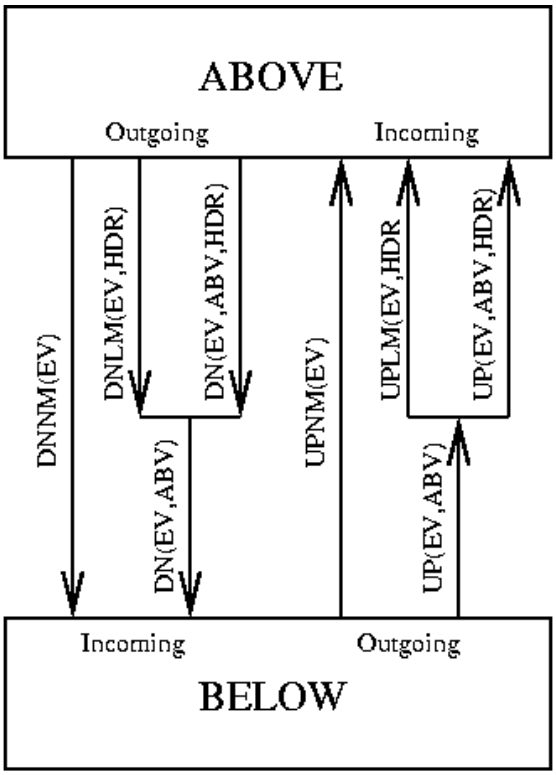

Figure 5: *Diagram of the 10 standard event handlers. Note that the ABOVE layer has a similar interface above it as the BELOW layer. Likewise with the interface beneath the BELOW layer.*

8 The Ensemble Security Architecture (by Ohad Rodeh)

This section describes the Ensemble security architecture. We believe that Ensemble completely supports the fortress security model. Only trusted, authorized members are allowed into the group. Once a member is allowed into a group, it is completely trusted. Ensemble is not secure against attacks from members that have been admitted into the group: any group member can break the protocols by sending bad messages.

The goal of our architecture is to secure group messages from tampering and eavesdropping. To this end, all group messages are signed and (possibly) encrypted. While it is possible to use public key cryptography for this task, we find this approach unacceptably expensive. Since all group members are mutually trusted, we share a symmetric encryption key, and a MAC [1] key among them. These keys are used to seal all group messages, making the seal/unseal operation very fast[2]. As a shorthand, we shall refer to the key-pair as the *group key*. Using a group key raises two challenges:

A rekeying mechanism: allowing secure replacement of the current group key once it is deemed insecure, or if there is danger that it was leaked to the adversary. Dissemination of the new key should be performed without relying on the old (compromised) group key.

Secure key agreement in a group: i.e., a protocol that creates secure agreement among group members on a mutual group key.

We focus on benign failures and assume that authenticated members will not be corrupted. Byzantine fault tolerant systems suffer from poor performance since they use costly protocols and make extensive use of public key cryptography. We believe that our failure model is sufficient for the needs of most practical applications.

The user may specify a security policy for an application. The policy specifies for each address[3] whether it is trusted or not. Each application maintains its own policy, it is up to Ensemble to enforce it and to allow only mutually trusted members into the same subgroup. A policy allows an application to specify the members that it trusts and exclude untrusted members from its subgroup.

8.1 Cryptographic Infrastructure

Our design supports the use of a variety of authentication and encryption mechanisms. Ensemble has been interfaced with the OpenSSL (see `http://www.openssl.org/`) cryptographic library, the PGP authentication engine, and the Kerberos centralized authentication system (this is out of date). By default, messages are signed using MD5, encrypted using RC4, and authentication is performed using PGP. Because these three functionalities are carried out independently any combination of supported authentication, signature, and encryption systems can be used. A future goal is to allow multiple systems to be supported concurrently. Under such a system, processes would be able to compare the systems they have support for and select any system that both have support for.

[1] MAC, Message Authentication Code. This is typically a keyed hash function.

[2] symmetric encryption/MAC is roughly 1000 times faster than equivalent public key operations.

[3] An Ensemble address is comprised of a set of identifiers, for example an IP address and a PGP principal name. Generally, an address includes an identifier for each communication medium the endpoint is using {UDP,TCP,MPI,ATM,..}.

8.2 Rekeying

Ensemble rekeying uses the notion of *secure channels*. A secure channel between endpoints p and q is essentially a symmetric encryption key k_{pq} agreed upon between p and q. This key is known only to p and q and is different than the group key. Whenever confidential information needs to be passed between p and q it is encrypted using k_{pq} and sent using Ensemble reliable point-to-point messaging.

The basic rekeying protocol supported uses a binary tree structure. In order to rekey the group, a complete binary tree spanning the group is created. Member 0 is the father of 1 and 2, 1 is the father of 3 and 4, etc.. The leader chooses a new key k_{new} and sends it securely to 1 and 2; member number 1 sends k_{new} securely down to 3 and 4, etc.. When a tree leaf receives a new key it sends up a clear-text acknowledgment. When acknowledgments reach the leader (0) it prompts the group for a view change in which the new key will be used.

k_{new} is disseminated confidentially using secure channels. We cannot use the old key to protect k_{new} since the old key is assumed to be compromised. Secure channels are created upon demand by Ensemble, they are then cached for future use. Creating a secure channel is a costly operation taking hundreds of milliseconds even on fast CPUs. It is performed in the background so as not to block the application.

Recently, we have added faster rekeying protocols to the system. A complete implementation of the dWGL algorithm has been added, in the form of several layers. There are two new algorithms rekey_dt, and rekey_diam. There are described in the reference manual.

8.3 A secure stack

The Security architecture is comprised of 5 layers:

Exchange: secure key agreement. This layer is responsible for securely handing the group key to new joining group components. Component leaders mutually authenticate and check authorization policies prior to handing the group key securely between them.

Encrypt: chain-encryption of all user messages.

Secchan: create and manage a cache of secure channels.

PerfRekey: handling common rekeying tasks. For example, after a new key has been disseminated to the group, acknowledgments must be collected from all group members.

Rekey_dt: Binary tree rekeying. Rekeying a group is very fast once secure channels have been setup. We logged an average rekey operation for a 20 member group at 100 milliseconds. Rekey_dt assumes that the Secchan and PerfRekey layers are in the stack.

The regular and secure Ensemble stacks are depicted in Figure 2. The Top and Bottom layer cap the stack from both sides. The membership layers compute the current set of live and connected machines, the Appl_top layer interfaces with the application and provides reliable send and receive capabilities for point-to-point and multicast messages. The RFifo layers provide reliable per-source fifo messaging. The Exchange and Rekey layers are related to the membership layers since the group key is a part of the view information. The Encrypt layer encrypts all user messages hence it is below the Appl_top layer.

Regular	security additions
Top	
	Exchange
	Rekey_dt
	PerfRekey
	Secchan
Gmp	
Top_appl	Interface to the application
	Encrypt
Rfifo	
Bottom	

Table 2: The Ensemble stack. On the left is the default stack that includes an application interface, the membership algorithm and a reliable-fifo module. To the right is a secure stack with the Exchange, Encrypt, Rekey_dt, and Secchan layers in place.

8.4 Security events

There are three security events to note:

- ERekey: By this event the application requests a Rekey operation.
- ESecureMsg: This event is used by the Rekey layer to send private messages to other processes. The Secchan layer catches this event and sends the message securely to its destination.
- ERekeyPrcl: this event is used in the communication between all rekeying layers.

The Vs_key field was added to the view state was to allow for group keys. It holds the current group key.

8.5 Using Security

Ensemble has three security properties:

1. Rekey: Add rekeying to the stack.
2. OptRekey: Use the dWGL algorithm for rekeying.
3. Auth: Authenticate all messages.
4. Privacy: Encrypt all user messages.

An application wishing for strong security should choose all of the above properties in its stack and perform a *Control Rekey* action once every several hours. Note that there are two flavors to application Rekey-ing:

- Rekey false: The default, as above.
- Rekey true: Cleanup prior to rekeying. For performance considerations, Ensemble keeps cached key-ing material and secure channels. These should be cleared up every couple of hours to prevent an adversary from using cryptanalysis to discover the group key.

An example command line, for application appl, with pgp user name James_Joyce:

```
appl -add_prop Auth -add_prop Privacy -key 01234567012345670123456701234567
     -pgp James_Joyce
```

In order to add authorization to the stack, thereby controlling which members are allowed to join a group, one must do:

```
  val policy_function : Addr.set -> bool
  val interface : Appl_intf.New.t

  let state = Layer.new_state interface in
  let state = Layer.set_exchange (Some policy_function) state in
  Appl.config_new_full state (ls,vs)
```

Instead of simply:

```
  Appl.config_new interface state (ls,vs)
```

Authorization is not linked to the Security architecture, regular stacks can perform authorization. Control of joining members is delegated to the group leader that checks its authorization list and allows/disallows join. Every view change the authorization list is checked and existing members that are not authorized are removed.

In practice, if an application changes its authorization list dynamically, it must perform a *P*rompt and a *R*ekey whenever such a change occurs.

8.6 Checking that things work

To check that PGP has been installed correctly, that Ensemble can talk to it without fault, and the cryptographic support is running correctly, one can use the armadillo demo program.

In order to set up PGP, one must create principals and corresponding public and private keys. These are installed by PGP in its local key repository. The basic PGP key-generation command is:

```
zigzag ~/ensemble/demo> pgp -kg
```

To work with the armadillo demo, you'll need to create principals in the group $o1, o2, ...$ Armadillo creates a set of endpoints, and then runs a test between them. To this end, the program has a "-n" flag that describes the number of endpoints to use. For example, the command line `armadillo -n 2 ...` tells armadillo that use a two members configuration. These members will have principal names $o1$ and $o2$ respectively.

To view the set of principals in the repository do:

```
zigzag ~/ensemble/demo> pgp -kv
pub   512/2F045569 1998/06/15 o2
pub   512/A2358EED 1998/06/15 o1
2 matching keys found.
```

To check that PGP runs correctly do:

```
zigzag ~/ensemble/demo> armadillo -prog pgp
PGP works
check_background
got a ticket
background PGP works
```

If something is broken, the PGP execution trace can be viewed using:

```
zigzag ~/ensemble/demo> armadillo -prog pgp  -trace PGP
```

If more information is required use the flags `-trace PGP1 -trace PGP2`. The default version of PGP that Ensemble works with is 2.6. If, however, you'd like to use a different version, set your environment variable $ENS\_PGP\_VERSION$ to the version number. Versions 5.0 and 6.5 are also supported.

To check that OpenSSL is installed correctly, one can do:

```
zigzag ~/ensemble/demo> armadillo -prog perf
```

For a wider scale test use the *exchange* test. This is a test that creates a set of endpoints, with principal names: *o1, o2, ...*, and merges them securely together into one group. Each group merge requires that group-leaders properly authenticate themselves using PGP. The test is started with all members in components containing themselves, and ends when a single secure component is created. Note that it will keep running until reaching the timeout. The timeout is set by default to 20 seconds. To invoke the test do:

```
zigzag ~/ensemble/demo> armadillo -prog exchange -n 2 -real_pgp
```

If something goes wrong, a trace of the authentication protocol is available through `-trace EXCHANGE`.

The `-real_pgp` flag tells armadillo not to simulate PGP. Simulation is the default mode for armadillo, since we use it to test communication protocol correctness.

To check that rekeying works do:

```
zigzag ~/ensemble/demo> armadillo -prog rekey  -n 5
```

To test security with two separate processes do the following:

```
zigzag ~/ensemble/demo> gossip &
zigzag ~/ensemble/demo> mtalk -key 1111222233334444111122223333444
                  -add_prop Auth -pgp o1
zigzag ~/ensemble/demo> mtalk -key 01234567012345670123456701234567
                 -add_prop Auth -pgp o2
```

The two mtalk processes should authenticate each other and merge.

The three command line arguments specify:

- `-key 1111222233334444111122223333444` : The initial security key of the system. Should be a 16 byte string.
- `-add_prop Auth`: Add the authentication protocol. Otherwise, stacks with different keys will not be able to merge.
- `-pgp o1`: Specify the principal name for the system.

8.7 Using security from HOT and EJava

The security options have been added to the HOT interface. For a demonstration program look at hot_sec_test.c in the hot subdirectory. The only steps one needs to make are: (1) Set the program's principal name (2) Set the security bit. Both of these options are specified in the join-options structure. For example, in hot_sec_test.c:

```
static void join(
 int i,
 char **argv
)
  state *s ;
  s = (state *) hot_mem_Alloc(memory, sizeof(*s)) ;
  memset(s,0,sizeof(*s)) ;

  s->status = BOGUS;
  s->magic = HOT_TEST_MAGIC;

  ...

  strcpy(s->jops.transports, "UDP");
  strcpy(s->jops.group_name, "HOT_test");

  ...

  sprintf(s->jops.princ, "Pgp(o%d)",i);
  s->jops.secure = 1;

  ...

  /* Join the group.
   */
  err = hot_ens_Join(&s->jops, &s->gctx);
  if (err != HOT_OK)
    hot_sys_Panic(hot_err_ErrString(err));
```

EJava is interfaced with HOT, so they share a similar interface. Note that the outboard mode, supported by both interface is **insecure**. The messages passing on the TCP connection between the client and server are neither MACed nor encrypted. Therefore, they can be used securely only when situated on a single machine.

Part II
The Ensemble Protocols

9 Layers and Stacks

We document a subset of the Ensemble layers and stacks (compositions of layers) in this section. This documentation is intended to be largely independent of the implementation language. They are currently listed in order, bottom-up, of their use in the VSYNC layer.

Each layer (or stack) has these items in its documentation:

9.1 ANYLAYER

The name of the layer follwed by a general description of its purpose.

Protocol

A description of the protocol implemented by the layer.

Parameters

- The list of parameters required to initialize the layer, along with descriptions of their purpose.
- [**should also specify reasonable values**]
- If a layers takes no arguments, the documentation specifies "None."

Properties

- A list of informal properties of the layer.

Notes

- General notes about the layer.

Sources

The source files for the ML implementation of the layer.

Generated Events

A list of event types generated by the layer. In the future, this field will contain more information, such as what event types are examined by the layer (instead of being blindly passed on). Hopefully, this information will eventually be generated automatically.

Testing

- Information about the status of the layer regarding testing.
- Testing information should always be documented: if the layer has not been tested, that should be stated.
- What testing has been completed on the layer (along with version information).
- What infrastracture is in place for testing the layer.
- Known bugs for a layer are listed in the ML source code.

9.2 CREDIT

This layer implements a credit based flow control.

Protocol

On initialization, sender informs receivers how many credits it wants to keep in stock. Receivers sends credits whenever it finds that the sender is low on credits, either explicitly through a sender's request or implicitly through its local accounting. A credit is one time use only. Sender is allowed to send a message only if it has a credit available. If the sender does not have a credit, the message is buffered. Buffered messages are sent when new credits arrive. Credits are piggybacked to data messages whenever there is an opportunity of doing so to save bandwidth.

Parameters

- rtotal: the total number of credits that this member can give out. Should be set according to the number of receive buffers that the machine the member is running has.
- ntoask: the number of credits that this member likes to keep in stock.
- whentoask: the threshold number of credits remaining at sender before the receiver consider sending out more credits.
- pntoask: like ntoask for piggyback style of credit giving.
- pwhentoask: like nwhentoask for piggyback style of credit giving.
- sweep: frequency at which periodic sweep routine, which give out credits to senders, should run.

Notes

- Future implementation should support dynamic credit adjustment.
- Alternative flow control layers include RATE and WINDOW.

Sources

layers/credit.ml

Last updated: Fri Mar 29, 1996

9.3 RATE

This layer implements a sender rate based flow control. Multicast messages from each sender are sent at a rate not exceeding some prescribed value.

Protocol

All the messages to be sent are buffered initially. Buffered messages are sent on periodic timeouts that are set based on the sender's rate.

Parameters

- rate_n, rate_t: the pair determines the rate. At most **rate_n** messages are allowed to sent over any time period of **rate_t**. This is ensured by having two consecutive messages sent with a inter-send time of at least $(rate\_t/rate\_n)$ apart.

Notes

- Future implementation should support dynamic rate adjustment.
- Alternative flow control layers include CREDIT and WINDOW.

Sources

layers/rate.ml

This layer and its documentation were written by Takako Hickey.

9.4 BOTTOM

Not surprisingly, the BOTTOM layer is the bottommost layer in a Ensemble protocol stack. It interacts directly with the communication transport by sending/receiving messages and scheduling/handling timeouts. The properties implemented are all *local* to the protocol stack in which the layer exists: ie., a (dn)Fail event causes failed members to be removed from the local view of the group, but no failure message to be sent out–it is assumed that some other layer actually informs the other members of the failure.

Protocol

None

Parameters

- None

Properties

- Requires messages be appropriately fragmented for the transport in use.
- **Dn(ETimer)**{time} events cause an alarm to be scheduled with the transport so that an **Up(ETimer)** event is later delivered some time after *time*.
- **Dn(EBlock)**, **Dn(EView)**, **Dn(EStable)**, and **Dn(EFail)** events cause an **Up(EBlock)**, **Up(EView)**, **Up(EStable)**, and **Up(EFail)** event (respectively) to be locally "bounced" up the protocol stack. No communication results from these events.
- In addition, **Dn(EFail)** events cause further Send and Cast messages from the failed members to be dropped.
- **Dn(EView)** events do not affect the membership in the current protocol stack. The view in the resulting **Up(EView)** event is merely a proposal for the next view of the group. (It is expected that a new protocol stack will be created for that view.)
- **Dn(ESend)**, **Dn(ECast)** events cause messages to be sent (unreliably) to other members in the group. The resulting **Up(ESend)** and **Up(ECast)** events are delivered with the origin field set with the rank of the sender and the time field set with the time that the messages was received (according to the transport).
- **Dn(EMerge)**, **Dn(EMergeDenied)**, and **Dn(EMergeGranted)** causes messages to be sent (unreliably) to members outside of the group. These result in **Up(EMergeRequest)**, **Up(EMergeDenied)**, and **Up(EMergeGranted)** messages at the destination, respectively.
- **Dn(ELeave)** events disable the transport instance and bounce up an **Up(EExit)** event. No further events are delivered after the **Up(EExit)**. [**currently, this may not be true**]

Sources

layers/bottom.ml

Generated Events

Up(EBlock)
Up(ECast)
Up(EExit)
Up(EFail)
Up(EStable)
Up(EMergeDenied)
Up(EMergeGranted)
Up(EMergeRequest)
Up(ESend)
Up(ESuspect)
Up(ETimer)
Up(EView)

Testing

- see the VSYNC stack

9.5 CAUSAL

The CAUSAL layer implements causally order multicast. It assumes reliable, FIFO ordered reliable messaging from layers below.

Protocol

The protocol has two versions: full and compressed vectors. First, we explain the simple version which uses full vectors. Then, we explain how these vectors are compressed.

Each outgoing message is appended with a *causal vector*. This vector contains the last causally delivered message from each member in the group. Each received message is checked for deliverability. It may be delivered only if all messages which it causally follows, according to its causal vector, have been delivered. If it is not yet deliverable, it is delayed in the layer until delivery is possible. A view change erases all delayed messages, since they can never become deliverable.

Causal vectors become large with the group size, so they must be compressed in order for this protocol to scale. The compression we use is derived from the Transis system. We demonstrate with an example: assume the membership includes three processes p, q and r. Process p sends message $m_{p,1}$, q sends $m_{q,1}$, causally following $m_{p,1}$ and r sends $m_{r,1}$ causally following $m_{q,1}$. The causal vector for $m_{r,1}$ is $[1|1|1]$. There is redundancy in the causal vector since it is clear that $m_{r,1}$ follows $m_{r,0}$. Furthermore, since $m_{q,1}$ follows $m_{p,1}$ we may omit stating that $m_{r,1}$ follows $m_{p,1}$. To conclude, it suffices to state that $m_{r,1}$ follows $m_{q,1}$. Using such optimizations causal vectors may be compressed considerably.

Sources

layers/causal.ml

Testing

- The CHK_CAUSAL protocol layer checks for CAUSAL delivery.

This layer and its documentation were written by Ohad Rodeh.

9.6 ELECT

This layer implements a leader election protocol. It determines when a member should become the coordinator. Election is done by delivering an **Dn(EElect)** event at the new coordinator.

Protocol

When a member suspects all lower ranked members of being faulty, that member elects itself as coordinator.

Parameters

- None

Properties

- **Up(ESuspect)** events may cause a **Dn(EElect)** event to be generated.

Sources

layers/elect.ml

Generated Events

Dn(EElect)

Testing

- see also the VSYNC stack

9.7 ENCRYPT

This layer encrypts application data for privacy. Uses keys in the view state record. Authentication needs to be provided by the lower layers in the system. The protocol headers are not encrypted. This layer must reside above FIFO layers for sending and receiving because it uses encryption contexts whereby the encryption of a message is dependent on the previous messages from this member. These contexts are dropped at the end of a view. A smarter protocol would try to maintain them, as they improve the quality of the encryption.

Protocol

Does chained encryption on the message payload in the **iov** field of events. Each member keeps track of the encryption state for all incoming and outgoing point-to-point and multicast channels. Messages marked **Unreliable** are not encrypted (these should not be application messages).

Parameters

- None

Properties

- Guarantees (modulo encryption being broken) that only processes that know the shared group key can read the contents of the application portion of data messages.
- Requires FIFO ordering on point-to-point and multicast messages.

Sources

layers/encrypt.ml

Generated Events

None

Testing

- see the VSYNC stack

9.8 HEAL

This protocol is used to merge partitions of a group.

Protocol

The coordinator occasionally broadcasts the existence of this partition via **Dn(EGossipExt)** events. These are delivered unreliably to coordinators of other partitions. If a coordinator decides to merge partitions, then it prompts a view change and inserts the name of the remote coordinator in the **Up(EBlockOk)** event. The INTER protocol takes over from there. Merge cycles are prevented by only allowing merges to be made from smaller view id's to larger view id's.

Parameters

- heal_wait_stable : whether or not to wait for a first broadcast message to become stable before starting the protocol. This ensures that all the members are in the group.

Properties

- [**TODO:**]

Sources

layers/heal.ml

Generated Events

Up(EPrompt)
Dn(EGossipExt)

Testing

- see the VSYNC stack

9.9 INTER

This protocol handles view changes that involve more than one partition (see also INTRA).

Protocol

Group merges are the more complicated part of the group membership protocol. However, we constrain the problem so that:

- Groups cannot be both merging and accepting mergers at the same time. This eliminates the potential for cycles in the "merge-graph."
- A view (i.e. view_id) can only attempt to merge once, and only if no failures have occured. Each merge attempt is therefore uniquely identified by the view_id of the merging group. Note also that by requiring no failures to have occured for a merge to happen, this prevents a member from being failed in one view and then reappearing in the next view. There has to be an intermediate view without the failed member: this is a desirable property.

The merge protocol works as follows:

1. The merging coordinator blocks its group,
2. The merging coordinator sends a merge request to the remote group's coordinator.
3. The remote coordinator blocks its group,
4. The remote coordinator installs a new view (with the mergers in it) and sends the view to the merging coordinator (through a merge-granted message).
5. The merging coordinator installs the view in its group.

If the merging coordinator times out on the merged coordinator then it immediately installs a new view in its partition (without the other members even finding out about the merge attempt).

Parameters

- None

Properties

- When another partition is merging, a View message is also sent to the coordinator of the merging group, which then forwards the message to the rest of its group.
- Requires that **Dn(EMerge)** events only be delivered by the original coordinator of views (in which no failures have yet occured). Otherwise, the partition should first form a new view and then attempt the merge.
- **Dn(EMerge)** causes a **Dn(EMerge)** event to be delivered to the layer below. This will be replied with either an **Up(EView)**, **Up(EMergeFailed)**, or **Up(EMergeDenied)** event, depending on the outcome of the merge attempt.
- **Up(EMergeRequest)**'s are only delivered at the coordinator. And only if the group is not currently blocking and only if the mergers list does not contain members that are/were in this view or in previous merge requests in this view.

Sources

layers/inter.ml

Generated Events

Dn(EMerge)
Dn(EMergeDenied)
Dn(ESuspect)

Testing

- see the VSYNC stack

9.10 INTRA

This layer manages group membership within a view (see also the INTER layer). There are three related tasks here:

- Forwarding of group membership events to the rest of the group (without INTRA, normally **Dn(EView)** and **Dn(EFail)** events have only local effect).
- Filtering of group membership events from remote members (for example, when two other group members think they are the coordinator and fail each other, the INTRA layer choose one of them and ignores the other member).
- Determining the view_id of the following view.

Protocol

This is a relatively simple group membership protocol. We have done our best to resist the temptation to "optimize" special cases under which the group is "unnecessarily" partitioned. We also constrain the conditions under which operations such as merges can occur. The implementation does not "touch" any data messages: it only handles group membership changes. Furthermore, this protocol does not use any timeouts.

Views and failures are forwarded via broadcast to the rest of the members. Other members accept the view/failure if they are consistent with their current representation of the group's state. Otherwise, the view/failure message is dropped and the sender is suspected of being problematic.

Parameters

- None

Properties

- **Dn(EView)** events are passed on to the layer below. They also cause a View message to be broadcast to the other members. On receipt of this View message, the other members either accept it (and deliver a **Dn(EView)** event to layer below) or mark the sender of the View as problematic, and possibly deliver a **Dn(ESuspect)** event to the layer below.
- Requires FIFO, atomic broadcast delivery from layers below.
- **Dn(EFail)** events are passed on to the layer below. They also cause a Fail message to be broadcast to the other members. On receipt of this Fail message, the other INTRA instances will either accept it (and deliver a **Dn(EFail)** event to the layer beneath them) or mark the sender of the Fail message as problematic, and possibly deliver an **Dn(ESuspect)** event to the layer below.
- View and Fail messages from a particular coordinator are delivered in FIFO order to the members.
- Not all members may see same set of **Up(EFail)** events. However, the set of failed members grows monotonicly with each failure notification.

Sources

layers/intra.ml

Generated Events

Dn(ECast)
Dn(EFail)
Dn(ESuspect)
Dn(EView)

Testing

- see the VSYNC stack

9.11 LEAVE

This protocol has two tasks. (1) When a member really wants to leave a group, the LEAVE protocol tells the other members to suspect this member. (2) The leave protocol garbage collects old protocol stacks by initiating a **Dn(ELeave)** after getting an **Up(EView)** and then getting an **Up(EStable)** where everything is marked as being stable.

Protocol

Both protocols are simple.

For leaving the group, a member broacasts a Leave message to the group which causes the other members to deliver a **Dn(ESuspect)** event. Note that the other members will get the Leave message only after receiving all the prior broadcast messages. This member should probably stick around, however, until these messages have stabilized.

Garbage collection is done by waiting until all broadcast message are stable before delivering a local **Dn(ELeave)** event.

Parameters

- leave_wait_stable : whether or not to wait for the leave announcment to become stable before leaving

Properties

- [**TODO:**]

Sources

layers/leave.ml

Generated Events

Dn(ELeave)

Testing

- see the VSYNC stack

9.12 MERGE

This protocol provides reliable retransmissions of merge messages and failure detection of remote coordinators when merging.

Protocol

Simple retransmission protocol. A hash table is used to detect copied merge requests, which are dropped.

Parameters

- merge_sweep: how often to retransmit merge requests
- merge_timeout: how long before timing out on merges

Properties

- **Dn(EMerge)**, **Dn(EMergeGranted)**, and **Dn(EMergeDenied)** events are buffered for later retransmission.
- **Up(EMergeRequest)**, **Up(EMergeGranted)**, and **Up(EMergeDenied)** events are filtered so that each event is delivered at most once by this layer (i.e., so that retransmissions are dropped).
- After *timeout* time (a parameter listed above) an **Up(EMergeFailed)** event is delivered with the problems field set to be the contact of the **Dn(EMerge)** (only) event. (It is assumed that the merge process will normally be complete before this timeout occurs.)

Notes

- Removal of this protocol layer only makes the merges unreliable, and stops the failure detection of the new coordinator.

Sources

layers/merge.ml

Generated Events

Up(ESuspect)
Dn(EMerge)
Dn(ETimer)

Testing

- see the VSYNC stack

9.13 MFLOW

This layer implements window-based flow control for multicast messages. Multicast messages from each sender are transmitted only if the number of send credit left is greater than zero. The protocol attempts to avoid situations where all recievers send credit at the same time, so that a sender is not flooded with credit messages.

Protocol

Whenever the amount of send credits drops to zero, messages are buffered without being sent. On receipt of acknowledgement credit, the amount of send credits are recalculated and buffered messages are sent based on the new credit.

Parameters

- mflow_window : the maximum amount on unacknowledged messages or the size of the window.
- mflow_ack_thresh : The acknowledge threshold. After receiving this number of bytes of data from a sender, the receiver acknowledged previous credit.

Properties

- This protocol bounds the number of unrecieved multicast messages a member has sent.
- The amount of received credits are initialized to different values for avoiding many members sending back acknowledge at the same time.
- This protocol requires reliable multicast and point-to-point properties from underlying protocol layers.

Notes

- As opposed to most of the Ensemble protoocols, this protocol implements flow control on bytes and not on messages. It only considers the data in the application payload portion of the message (the **iov** field of the event).
- Because of the EBlockOk events, this layer needs to be below the broadcast stability layer.

Sources

layers/mflow.ml

Testing

- Some testing has been carried out.

This layer and its documentation were written with Zhen Xiao.

9.14 MNAK

The MNAK (Multicast NAK) layer implements a reliable, agreed, FIFO-ordered broadcast protocol. Broadcast messages from each sender are delivered in FIFO-order at their destinations. Messages from live members are delivered reliably and messages from failed members are retransmitted by the coordinator of the group. When all failed members are marked as such, the protocol guarantees that eventually all live members will have delivered the same set of messages.

Protocol

Uses a negative acknowledgment (NAK) protocol: when messages are detected to be out of order (or the *NumCast* field in an **Up(EStable)** event detects missing messages), a NAK is sent. The NAK is sent in one of three ways, chosen in the following order:

1. Pt2pt to the sender, if the sender is not failed.
2. Pt2pt to the coordinator, if the reciever is not the coordinator.
3. Broadcast to the rest of the group if the receiver is the coordinator.

All broadcast messages are buffered until stable.

Parameters

- mnak_allow_lost : boolean that determines whether the MNAK layer will check for lost messages. Lost messages are only possible when using an inaccurate stability protocol.

Properties

- Requires stability and *NumCast* information (equivalent to that provided by the STABLE layer).
- **Dn(ECast)** events cause **Up(ECast)** events to be delivered at all other members (but not locally) in the group in FIFO order.
- **Up(EStable)** events from the layer below cause stable messages to be garbage collected and may cause NAK messages to be sent to other members in the group.

Sources

layers/mnak.ml

Generated Events

Dn(ECast)
Dn(ESend)

Testing

- The CHK_FIFO protocol layer checks for FIFO safety conditions.
- The FIFO application generates bursty communication in which a token traces its way through each burst. If a reliable communication layer drops the token, communication comes to an abrupt halt. This is intended to capture the liveness conditions of FIFO layers.
- see also the VSYNC stack

9.15 OPTREKEY

This layer is part of the dWGL suite. Together with RealKeys, it implements the dWGL rekeying algorithm. The specific task performed by OptRekey is computing the new group keygraph.

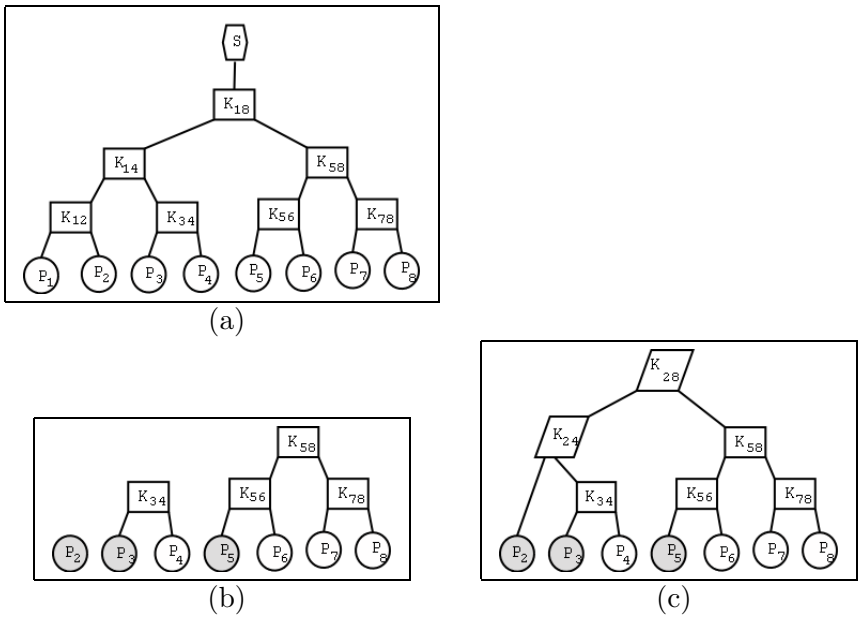

Figure 6: The effect of leave on a group key-graph of a group G of eight members. (1) The initial keygraph. (2) The tree after member p_1 leaves. (3) The merged tree.

Briefly, a keygraph is a graph where all group members form the leaves, and the inner nodes are shared sub-keys. A member knows all the keys on the route from itself to the root. The top key is the group key, and it is known by all members. For example, Figure 6(a) depicts a group G of eight members $\{p_1 \dots p_8\}$ and their subkeys. When a member leaves the group, all the keys known to it must be discarded. This splits a group into a set of subtrees. Figure 6(b) shows G after member p_1 has left. In order to re-merge the group keygraph, the subtrees should be merged. This can be seen in Figure 6(c). A subleader is the leader of a subtree. In our example, member p_2 is the leader of $\{p_2\}$, p_3 is the leader of $\{p_3, p_4\}$, and p_5 is the leader of $\{p_5, p_6, p_7, p_8\}$.

Protocol

This layer is activated upon a Rekey action. The leader receives an **ERekeyPrcl** event, and starts the OptRekey protocol. Typically, a Rekey will follow a join or a leave. Hence, the group keygraph is initially fragmented. This layer's task is to remerge it. The protocol employed is as follows:

1. The leader multicasts *Start*.
2. Subleaders send their keygraphs to the leader.
3. The leader computes an optimal new keygraph.
4. The leader multicasts the new keygraph.

5. Members receive the keygraph and send it up using a **ERekeyPrcl** event to the RealKeys layer.

An optimal keygraph is complex to compute, an auxiliary module is used for this task. Note that OptRekey is designed so that only subleaders participate. In the normal case, where a single member joins or leaves, this will include $log_2 n$ members.

It is possible that a Rekey will be initiated even though membership hasn't changed. This case is specially handled, since it can be executed with nearly no communication.

Properties

- Requires VSYNC properties.

Sources

layers/optrekey.ml
layers/util/tree.ml,mli
layers/type/tdefs.ml,mli

Generated Events

Dn(ECast)
Dn(ESend)

Testing

- The armadillo program (in the demo subdirectory) tests the security properties of Ensemble.

9.16 PERFREKEY

This layer is responsible for common management tasks related to group rekeying. Above PerfRekey, a rekeying layer is situated. At the time of writing there are four options: Rekey, RealKeys+OptRekey, Rekey_dt, and Rekey_diam. The Rekey layer implements a very simple rekeying protocol, RealKeys and OptRekey layers together implement the dWGL protocol. Rekey_dt implements a dynamic-tree based protocol, and Rekey_diam uses a diamond-like graph.

Protocol

The layer comes into effect when a Rekey operation is initiated by the user. It is bounced by the Bottom layer as a **Rekey** event and received at PerfRekey. From this point, following protocol is used:

1. The **Rekey** action is diverted to the leader.
2. The leader initiates the rekey sequence by passing the request up to Rekey/OptRekey/Rekey_dt/Rekey_d
3. Once rekeying is done, the members pass a **RekeyPrcl** event with the new group key back down.
4. PerfRekey logs the new group key. A tree spanning the group is computed through which acks will propagate. The leaves sends Acks up the tree.
5. When Acks from all the children are received at the leader, it prompts the group for a view change.

In the upcoming view, the new key will be installed.

Another rekeying flavor includes a *Cleanup* stage. Every couple of hours, the set of cached secure channels, and other key-ing material should be removed. This prevents an adversary from using cryptanalysis to break the set of symmetric keys in use by the system. To this end, PerfRekey supports an optional cleanup stage prior to actual rekeying. This is a sub-protocol that works as follows:

1. The leader multicasts a *Cleanup* message.
2. All members remove all their cached key-material from all security layers. A **ERekeyCleanup** event is sent down to Secchan, bounced up to Rekey/OptRekey+RealKeys/.., and bounced back down to PerfRekey.
3. All members send *CleanupOk* to the leader through the Ack-tree.
4. When the leader receives *CleanupOk* from all the members, it starts the Rekey protocol itself.

By default, cleanup is perform every 24hours. This is a settable parameter that the application can decide upon.

Rekeying may fail due to member failure or due to a merge that occurs during the execution. In this case, the new key is discarded and the old key is kept. PerfRekey supports persistent rekeying: when the 24hour timeout is over, a rekey will ensue no-matter how many failures occur.

The Top layer checks that all members in a view a trusted. Any untrusted member is removed from the group through a **Suspicion** event. Trust is established using the Exchange layer, and the user access control policy.

Properties

- Requires VSYNC properties.
- Guarantees during a view change, either all members switch to the new shared key or none of them do.

Parameters

- perfrekey_ack_tree_degree: determines the degree of the Ack tree. The default is 6.
- perfrekey_sweep: determines the compulsory cleanup period.

Sources

layers/perfrekey.ml

Generated Events

EPrompt
ERekeyPrcl
Dn(ECast)
Dn(ESend)

Testing

- The armadillo program (in the demo subdirectory) tests the security properties of Ensemble.

9.17 PRIMARY

Detect primary partition in a group. Usually a primary partition has the majority of members or holds some important resources.

Protocol

Upon **Up(EInit)** event, a member sends a message to the coordinator, claiming that it is in the current view. When a view has the majority of members, its coordinator prompts a view change to make itself the primary partition if it is not yet. When a new view is ready, it decides whether it is primary and mark it as so.

Parameters

- primary_quorum: how many servers (non-client member) are needed to form the primary partition.

Properties

- Guarantees no two primary partitions can have the same logical timestamp.
- Optimal in the normal case: no addtional view change is necessary.
- This protocol requires group membership management from underlying protocol layers.

Sources

layers/primary.ml

Generated Events

Dn(EPrompt)
Dn(ESend)

Testing

- TODO

This layer and its documentation were written with Zhen Xiao.

9.18 PT2PT

This layer implements reliable point-to-point message delivery.
[**TODO: finish this documentation**]

Parameters

- pt2pt_sweep : how often to retransmit messages and send out acknowledgments
- pt2pt_ack_rate : determines how many messages will be received before an acknowledgement is generated.
- pt2pt_sync : boolean determining if point-to-point messages should be synchronized with view changes

Testing

- see the VSYNC stack

9.19 PT2PTW

This layer implements window-based flow control for point to point messages. Point-to-point messages from each sender are transmitted only if the window is not yet full.

Protocol

Whenever the amount of send credits drops to zero, messages are buffered without being sent. On receipt of acknowledgement credit, the amount of send credits are recalculated and buffered messages are sent based on the new credit. Acknowledgements are sent whenever a speicified threshhold is passed.

Parameters

- pt2ptw_window : the maximum amount on unacknowledged messages or the size of the window.
- pt2ptw_ack_thresh : The acknowledge threshold. After receiving this number of bytes of data from a sender, the receiver acknowledges previous credit.

Properties

- This protocol bounds the number of unrecieved point-to-point messages a member can send.
- This protocol requires reliable point-to-point properties from underlying protocol layers.

Notes

- As opposed to most of the Ensemble protocols, this protocol implements flow control on bytes and not on messages. It only considers the data in the application payload portion of the message (the **iov** field of the event).

Sources

layers/pt2ptw.ml

Testing

- Some testing has been carried out.

Last updated: March 21, 1997

,

9.20 PT2PTWP

This layer implements an adaptive window-based flow control protocol for point-to-point communication between the group members.

In this protocol the receiver's buffer space is shared between all group members. This is accomplished by dividing the receiver's window among the senders according to the bandwidth of the data being received from each sender. Such way of sharing attempts to minimize the number of ack messages, i.e. to increase message efficiency.

Protocol

In the following, the term acknowledgement is used with the meaning of flow control protocols and not that of reliable communication protocols.

This protocol uses *credits* to measure the available buffer space at the receiver's side. Each sender maintains a *window* per each destination, which is used to bound the unacknowledged data a process can send point-to-point to the given destination. For each message it sends, the process deducts a certain amount of credit based on the size of the message. Messages are transmitted only if the sender has enough credit for them. Otherwise, messages are buffered at the sender.

A receiver keeps track of the amount of unacknowledged data it has received from each sender. Whenever it decides to acknowledge a sender, it sends a message containing new amount of credit for this sender. On receipt of an acknowledgement message, sender recalculates the amount of credit for this receiver, and the buffered messages are sent based on the new credit.

The receiver measures the bandwidth of the data being received from each sender. It starts with zero bandwidth, and adjusts it periodically with timeout *pt2ptwp_sweep*.

On receipt of a point-to-point message, the receiver checks if the sender has passed threshold of its window, i.e. if the amount of data in point-to-point messages received from this sender since the last ack was sent to it has exceeded a certain ratio, *pt2ptwp_ack_thresh*, of the sender's window. If it is, an ack with some credit has to be sent to the sender. In order to adjust processes' windows according to their bandwidth, the receiver attempts to steal some credit from an appropriate process and add it to the sender's window. The receiver looks for a process with maximal $\frac{window}{\sqrt{bandwidth}}$ ratio, decreases its window by certain amount of credit and increases the window of the sender appropriately. Then the receiver sends the sender ack with the new amount of credit. When the process from which the credit was stolen passes theshold of its new, smaller window, the receiver sends ack to it.

Parameters

- pt2ptwp_window : size of the receiver's window, reflects the receiver's buffer space.
- pt2ptwp_ack_thresh : the ratio used by the receiver while deciding to acknowledge senders.
- pt2ptwp_min_credit : minimal amount of credit each process must have.

- pt2ptwp_bw_thresh : credit may be stolen for processes with greater bandwidth only.
- pt2ptwp_sweep : the timeout of periodical adjustment of bavdwidth.

Properties

- This protocol requires reliable point-to-point properties from underlying protocol layers.

Notes

- As opposed to most of the Ensemble protocols, this protocol implements flow control on bytes and not on messages. It only considers the data in the application payload portion of the message (the **iov** field of the event).

Sources

layers/pt2ptwp.ml

Testing

- Correctness and performance testing has been carried out.

9.21 REALKEYS

This layer is part of the dWGL suite. Together with OptRekey is implements the dWGL protocol. This layer's task is to actually perform the instructions passed to it from OptRekey, generate and pass securely all group subkeys, and finally the group key.

Protocol

When a Rekey operation is performed a complex set of layers and protocols is set into motion. Eventually, each group member receives a new keygraph and a set of instructions describing how to merge its partial keytree with the rest of the group keytrees to achieve a unified group tree. The head of the keytree is the group key.

The instructions are implemented in several stages by the subleaders:

1. Choose new keys, and send them securely to peer subleaders using secure channels.
2. Get new keys through secure channels. Disseminate these keys by encrypting them with the top subtree key, and sending pt-2-pt to the leader.
3. When the leader gets all 2nd stage messages, it bundles them into a single multicast and sends to the group.
4. A member p that receives the multicast, extracts the set of keys it should know. Member p creates an **ERekeyPrcl** event with the new group key attached. The event it send down to PerfRekey notifing it that the protocol is complete.

Properties

- Requires VSYNC properties.

Sources

layers/realkeys.ml
layers/type/tdefs.ml,mli

Generated Events

ESecureMsg
Dn(ECast)
Dn(ESend)

Testing

- The armadillo program (in the demo subdirectory) tests the security properties of Ensemble.

9.22 REKEY

This layers switches the group key upon request. There may be several reasons for switching the key:

- The key's lifetime has expired — it is now possible that some dedicated attacker has cracked it.
- The key has been compromised.
- Application authorization policies have changed and previously trusted members need to be excluded from the group.

This layer also relies on the Secchan layer to create secure channels when required. A secure channel is essentially a way to pass confidential information between two endpoints. The Secchan layer creates secure channels upon demand and caches them for future use. This allows the new group key to be disseminated efficiently and confidentially through the tree.

Protocol

When a member layer gets an **ERekeyPrcl** event, it sends a message to the coordinator to start the rekeying process. The coordinator generates a new key and sends it to its children using secure channels. The children pass it down the tree. Once a member receives the new key is passes it down to PerfRekey using an **ERekeyPrcl** event.

The PerfRekey layer is responsible for collecting acknowledgments from the members and performing a view change with the new key once dissemination is complete.

Parameters

- rekey_degree: The degree of the dissemination tree. By default it is 2.

Properties

- Guarantees during a view change, either all members switch to the new shared key or none of them do.

Sources

layers/rekey.ml

Generated Events

Dn(ECast)
Dn(ESend)

Testing

- The armadillo.ml file in the demo directory tests the security properties of Ensemble.

This layer was originally written by Mark Hadyen with Zhen Xiao. Ohad Rodeh later rewrote the security layers and related infrastructure.

9.23 REKEY_DT

This is the default rekeying layer. The basic data structure used is a tree of secure channels. This tree changes every view-change, therefore the name of the layer. Dynamic Tree REKEY.

The basic problem in obtaining efficient rekeying is the high cost of constructing secure channels. A secure channel is established using a two-way handshake using a Diffie-Hellman exchange. At the time of writing, a PentiumIII 500Mhz can perform one side of a Diffie-Hellman exchange (using the OpenSSL cryptographic library) in 40 milliseconds. This is a heavyweight operation.

To discuss the set of channels in a group, we shall view it as a graph where the nodes are group members, and the edges are secure channels connecting them. The strategy employed by REKEY_DT is to use a tree graph. When a rekey request is made by a user, in some view V, the leader multicasts a tree structure that uses, as much as possible, the existing set of edges.

For example, if the view is composed of several previous components, then the leader attempts to merge together existing key-trees. If a single member joins, then it is located as close to the root as possible, for better tree-balancing. If a member leaves, then the tree may, in the worst case, split into three pieces. The leader fuses them together using (at most) 2 new secure channels.

The leader chooses a new key and passes it to its children. The key is passed recursively down the tree until it reaches the leaves. The leaf nodes send acknowledgments back to the leader.

This protocol has very good performance. It is even possible, that a rekey will not require any new secure-channels. For example, in case of member leave, where the node was a tree-leaf.

Protocol

When a member layer gets an **ERekeyPrcl** event, it sends a message to the coordinator to start the rekeying process. The coordinator checks if the view is composed of a single tree-component. If not, it multicasts a *Start* message. All members that are tree-roots, sends their tree-structure to the leader. The leader merges the trees together, and multicasts the group-tree. It then chooses a new key and sends it down the tree.

Once a member receives the new key is passes it down to PerfRekey using an **ERekeyPrcl** event.

The PerfRekey layer is responsible for collecting acknowledgments from the members and performing a view change with the new key once dissemination is complete.

Sources

layers/rekey_dt.ml

Generated Events

Dn(ECast)
Dn(ESend)

Testing

- The armadillo.ml file in the demo directory tests the security properties of Ensemble.

9.24 REKEY_DIAM

This layer is closely related to REKEY_DT. It employs the same concept of a graph where the nodes are group members, and the edges are secure channels connecting them together.

REKEY_DIAM attempts to improve the efficiency of a rekey after member leave. The point is to support ACL changes efficiently. If the application decides to change its ACL and remove a member, then the group-key must be switched as quickly as possible. As long as the previous group key is in place, the untrusted member can eavesdrop on group messaging.

The key to a low-latency rekey protocol, is the elimination of costly Diffie-Hellman exchanges on it's critical path. A simple possibility is to arrange the members in a circle. If a member is removed, the circle is still one-connected, and confidential information can still pass through it. After the initial rekey, a *reconstruction* phase is initiated. During that phase, a new circle, connecting all surviving members is constructed. The problem with the circle structure, is that is has $O(n)$ diameter. Since the diameter determines the latency of the protocol, we require a structure that has logarithmic diameter. We use a diamond graph, see examples in Figure 7.

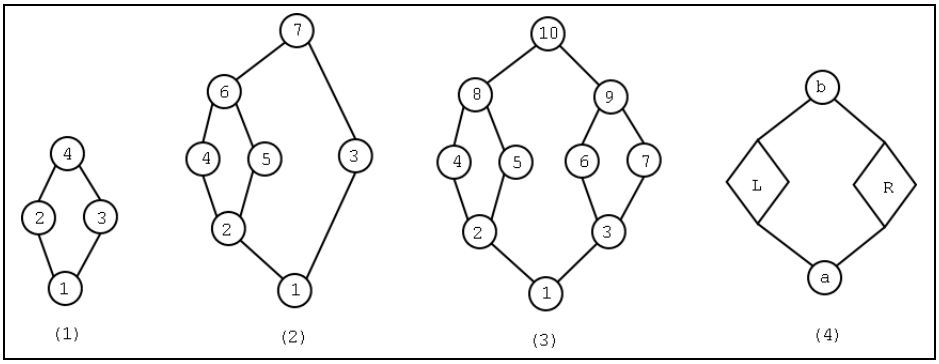

Figure 7: Examples for diamonds

The protocol handles merges, partitions, and diamond-graph balancing. It guarantees *very-low* latency for the case of member leave. We clocked it at four milliseconds on 20 member groups.

Sources

layers/rekey_diam.ml

9.25 SECCHAN

This layer is responsible for sending and receiving private messages to/from group members. Privacy is guaranteed through the creation and maintenance of *secure channels*.

A secure channel is, essentially, a symmetric key (unrelated to the group key) agreed upon between two members. This key is used to encrypt any confidential message sent between them. We allow layers above Secchan to send/receive confidential information using *SecureMsg* events. When a SecureMsg($dst, data$) event arrives at Secchan, a secure channel to member *dst* is created (if one does not already exist). Then, the *data* is encrypted using the secure channel key and reliably sent to *dst*.

This layer relies on an authentication engine - this is provided in system independent form by the Auth module. Currently, PGP is used for authentication. New random shared keys are generated by the Security module. The Security module also provides hashing and symmetric encryption functions. Currently RC4 is used for encryption and MD5 is used for hashing.

Protocol

A secure channel between members p and q is created using the following basic protocol:

1. Member p chooses a new random symmetric key k_{pq}. It creates a ticket to q that includes k_{pq} using the Auth module ticket facility. Essentially, Auth encrypts k_{pq} with q's public key and signs it using p's private key. Member p then sends the ticket to q.
2. Member q authenticates and decrypts the message, and sends an acknowledgment (*Ack*) back to p.

This two-phase protocol is used to prevent the occurrence of a *double channel*. By this we mean the case where p and q open secure channels to each other at the same time. We augment the Ack phase; q discards p's ticket if:

1. q has already started opening a channel to p
2. q has a larger[4] name than p.

Secchan also keeps the number of open channels, per member, below the *secchan_cache_size* configuration parameter. Regardless, a channel is closed if it's lifetime exceeds 8 hours (the setable *secchan_ttl* parameter). A two-phase protocol is used to close a channel. If members p and q share channel, assuming p created it, then p sends a *CloseChan* message to q. Member q responds by sending a *CloseChanOk* to p.

It typically happens that many secure channels are created simultaneously group wide. For example, in the first Rekey of a group. If we tear down all these channels exactly 8 hours from their inception, the group will experience an explosion of management information. To prevent this, we stagger channel tear down times. Upon creation, a channel's maximal lifetime is set to $8hours + Iseconds$ where I is a random integer in the range $[0\ ..secchan\_rand]$. *secchan_rand* is set by default to 200 seconds, which we view as enough.

[4]Polymorphic comparison is used here.

Properties

- Requires VSYNC properties.

Parameters

- secchan_cache_size: determines size of secure channel cache.
- secchan_ttl: Time To Live of a channel.
- secchan_rand: Used to stagger channel refresh times.
- secchan_causal_flag: for performance evaluations.

Sources

layers/secchan.ml
layers/msecchan.ml

Generated Events

EChannelList
ESecureMsg
Dn(ECast)
Dn(ESend)

Testing

- The armadillo program (in the demo subdirectory) tests the security properties of Ensemble.

9.26 SEQUENCER

This layer implements a sequencer based protocol for total ordering.

Protocol

One member of the group serves as the sequencer. Any member that wishes to send messages, send them point-to-point to the sequencer. The sequencer then delivers the message localy, and cast it to the rest of the group. Other members, as soon as they receive a cast from the sequencer, they deliver the message.

If a view change occurs, messages are tagged as unordered and are send as such. When the **Up(EView)** event arrives, indicating that the group has successfully been flushed, these messages are delivered in a deterministic order everywhere (according to the ranks of their senders, breaking ties using FIFO).

Parameters

- None

Properties

- Requires VSYNC properties.

Sources

layers/sequencer.ml

Generated Events

Dn(ECast)
Dn(ESend)

Testing

- **[TODO:]**

This layer and its documentation were written by Roy Friedman.

9.27 SLANDER

This protocol is used to share suspicions between members of a partition. This way, if one member suspects another member of being faulty, the coordinator is informed so that the faulty member is removed, even if the coordinator does not detect the failure. This ensures that partitions will occur even in the case of asymmetric network failures. Without the protocol, only when the coordinator notices the faulty member will the member be removed.

Protocol

The protocol works by broadcasting slander messages to other members whenever it recieves a new Suspect event. On the receipt of such a message, DnSuspect events are generated.

Parameters

- None

Properties

- If any member suspects another member of being faulty, all members will eventually suspect that member.
- **Up(ESuspect)** events may cause a Slander message to be generated.

Sources

layers/slander.ml

Generated Events

Dn(ESuspect)

Testing

- see also the VSYNC stack

This layer and its documentation were written by Zhen Xiao.

9.28 STABLE

This layer tracks the stability of broadcast messages and does failure detection. It keeps track of and gossips about an acknowledgement matrix, from which it occasionally computes the number of messages from each member that are stable and delivers this information in an **Dn(EStable)** event to the layer below (which will be bounced back up by a layer such as the BOTTOM layer).

Protocol

The stability protocol consists of each member keeping track of its view of an acknowledgement matrix. In this matrix, each entry, (A,B), corresponds to the number of member B's messages member A has acknowledged (the diagonal entries, (A,A), contain the number of broadcast messages sent by member A). The minimum of column A (disregarding entries for failed members) is the number of broadcast messages from A that are stable. The vector of these minimums is called the stability vector. The maximum of column A (disregarding entries of failed members) is the number of broadcast messages member A has sent that are held by at least one live member. The vector of the maximums is called the $NumCast$ vector [**there has got to be a better name**]. Occasionally, each member gossips its row to the other members in the group. Occasionally, the protocol layer recomputes the stability and $NumCast$ vectors and delivers them up in an **Dn(EStable)** event.

To prevent a message storm when members gossip their stability vectors, each member adds an initial time-delta to its timer. The deltas are spread between zero and `stable_spacing` based on member rank. For example, if there are 10 members, and suspect_spacing is set to 1 second, then the deltas for members zero through nine are: 0.0, 0.1, .., 0.9.

Parameters

- stable_sweep: how often to (1) gossip and (2) deliver stability (if it has changed)
- stable_explicit_ack: whether to request end-to-end acknowledgements for messages
- stable_spacing : the time-interval over which to spread periodic sending of stability vectors.

Properties

- Unless it is marked with the **Unreliable** option all DnCast events are counted by the STABLE layer and require eventual acknowledgement by the other members in the group in order to achieve stability.
- **Dn(EStable)** events from the stability layer have two extension fields set. The first is the **StableVect** extension, which is the vector of stability number of messages from each of the members in the group which are known to be stable. The second is the **NumCast** extension which is a vector with the number of broadcast messages each member in the group is known to have sent.
- **Dn(EStable)** events are never delivered before all live members have acknowledged at least the number of messages noted in the stability event. (safety)

- **Dn(EStable)** event will eventually be delivered after live members have acknowledged message **seqno** from member A, where the entry in the stable vector for member A is at least **seqno+1**. (liveness)
- The stability vectors in **Dn(EStable)** events from the STABLE layer are monotonically increasing.

Notes

- **NumCast** entries are not monotonicly increasing. For example, consider the case of member A broadcasting some messages (which are all dropped by the network), then broadcasting its gossip information (which are recieved), then failing. The other members may deliver some UpStable events with the number of known broadcasts from member A, in which the dropped broadcasts are counted. However, after the other members detect member A's failure, the **NumCast** entry for member A will be lowered to be the number of messages from A that the live members have recieved, which will be lower than when A was not failed.
- **Up(ECast)** events do not need to be acknowledged individually: an acknowledgment, **Ack(from,seqno)**, is taken to acknowledge all of the first **seqno** messages from the member with rank **from**.
- An attempt has been made to speed up stability detection during view changes by sending extra gossip messages when failures have occurred.

Sources

layers/stable.ml

Generated Events

Up(EStable)
Dn(ECast)
Dn(ETimer)

Testing

- see the VSYNC stack

9.29 SUSPECT

This layer regularly pings other members to check for suspected failures. Suspected failures are announce in a **Dn(ESuspect)** event to the layer below (which will be bounced back up by a layer such as the BOTTOM layer).

Protocol

Simple pinging protocol. Uses a sweep interval. On each sweep, Ping messages are broadcast unreliably to the entire group. Also, the number of sweep rounds since the last Ping was received from other members is checked and if it exceed the **max_idle** threshold then a **Dn(ESuspect)** event is generated.

To prevent a message storm when member's sweep timers expire, each member adds an initial time-delta to its sweep timer. The deltas are spread between zero and `suspect_spacing` based on member rank. For example, if there are 10 members, and suspect_spacing is set to 1 second, then the deltas for members zero through nine are: 0.0, 0.1, .., 0.9.

Parameters

- suspect_sweep : how often to Ping other members and check for suspicions
- suspect_max_idle : number of unacknowledged Ping messages before generating failure suspicions.
- suspect_spacing : the time-interval over which to spread periodic sending of suspicions.

Properties

- Suspicions are no guarantee that an actual failure has occured, only a guess.

Notes

- None

Sources

layers/suspect.ml

Generated Events

Dn(ESuspect)
Dn(ECast)
Dn(ETimer)

Testing

- see the VSYNC stack

9.30 SYNC

This layer implements a protocol for blocking a group during view changes. One member initiates the SYNC protocol by delivering a **Dn(EBlock)** event from above. Other members will receive an **Up(EBlock)** event. After replying with a **Dn(EBlockOk)**, the layers above the SYNC layer should not broadcast any further messages. Eventually, after all members have responded to the **Up(EBlock)** and all broadcast messages are stable, the member that delivered the **Dn(EBlock)** event will recieve an **Up(EBlockOk)** event.

Protocol

This protocol is very inefficient and needs to be reimplemented at some point. The Block request is broadcast by the coordinator. All members respond with another broadcast. When the coordinator gets all replies, it delivers up an **Up(EBlockOk)**

Parameters

- None

Properties

- Requires FIFO, reliable broadcasts with stability detection.
- Expects at most one **Dn(EBlock)** from above.
- Always delivers at most one **Up(EBlockOk)** event. Only delivers an **Up(EBlockOk)** if a **Dn(EBlock)** was recieved from above.
- When at least one member recieves a **Dn(EBlock)** event, all live members will eventually deliver an **Up(EBlock)** event.
- Expects at most one **Dn(EBlockOk)** event from above. Expects a **Dn(EBlockOk)** from above only if an **Up(EBlock)** event was previously delivered by this layer.
- Expects a **Dn(EBlock)** to the layers below will be replied with an **Up(EBlock)** from below.
- When all members have delivered a **Dn(EBlockOk)** event from above and all broadcast messages have been acknowledged (by non-failed members), eventually all members who delivered a **Dn(EBlock)** event will receive an **Up(EBlockOk)** event from this layer.

Sources

layers/sync.ml

Generated Events

Up(EBlockOk)
Dn(EBlock)
Dn(ECast)

Testing

- The CHK_SYNC protocol layer checks for SYNC safety conditions.
- see also the VSYNC stack

9.31 TOTEM

This layer implements the rotating token protocol for total ordering. (This is a variation on the protocol developed as part of the Totem project.)

Protocol

The protocol here is fairly simple: As soon as the stack becomes valid, the lowest ranked member starts rotating a token in the group. In order to send a message, a process must wait for the token. When the token arrives, all buffered messages are broadcast, and the token is passed to the next member. The token must be passed on even if there are no buffered messages.

If a view change occurs, messages are tagged as unordered and are send as such. When the **Up(EView)** event arrives, indicating that the group has successfully been flushed, these messages are delivered in a deterministic order everywhere (according to the ranks of their senders, breaking ties using FIFO).

Parameters

- None

Properties

- Requires VSYNC properties and local delivery.

Sources

layers/totem.ml

Generated Events

Dn(ECast)

Testing

- **[TODO:]**

This layer and its documentation were written by Roy Friedman.

9.32 WINDOW

This layer implements window-based flow control based on stability information. Multicast messages from each sender are sent only if the number of unacknowledged messages from the sender is smaller than the window.

Protocol

Whenever the number of unstable messages goes above the window, messages are buffered without being sent. On receipt of a stability update, the number of unstable messages are recalculated and buffered messages are sent as allowed by the window.

Parameters

- window_window : the window size in number of messages

Properties

- Requires stability information in the form of **Up(EStable)** events.

Notes

- Future implementation should support dynamic window adjustment.
- Performance with the WINDOW layer depends in part with the frequency of stability updates. The WINDOW flow control works the best when the frequency is based on the number of unstable messages rather than on periodic timeouts.
- Alternative flow control layers include RATE and CREDIT.

Sources

layers/window.ml

This layer and its documentation were written by Takako Hickey.

9.33 XFER

This protocol facilitates application based state-transfer. The view structure contains a boolean field `xfer_view` conveying whether the current view is one where state-transfer is taking place (`xfer_view = true`) or whether it is a regular view (`xfer_view = false`).

Protocol

It is assumed that an application initiates state-transfer after a view change occurs. In the initial view, `xfer_view = true`. In a fault free run, each application sends pt-2-pt and multicast messages, according to its state-transfer protocol. Once the application-protocol is complete, an `XferDone` action is sent to Ensemble. This action is caught by the Xfer layer, where each member sends a pt-2-pt message `XferMsg` to the leader. When the leader collects `XferMsg` from all members, the state-transfer is complete, and a new view is installed with the `xfer_view` field set to false.

When faults occur, and members fail during the state-transfer protocol, new views are installed with `xfer_view` set to `true`. This informs applications that state-transfer was not completed, and they can restart the protocol.

Notes

- This layer allows the application to choose the state-transfer protocol it wishes to use, the only constrain being the XferDone actions.
- In the normal case, (a fault free run) the protocol should take a single view to complete.

Parameters

- None

Properties

- Requires VSYNC properties.

Sources

layers/xfer.ml

9.34 ZBCAST

The ZBCAST layer implements a gossip-style probabilistically reliable multicast protocol. Unlike most other protocols in Ensemble, this protocol admits a small, but non-zero probability of message loss: a message might be garbage collected even though some operational member in the group has not received it yet. We found that doing so can offer dramatic improvements in the performance and scalability of the protocol.

Protocol

This protocol is composed of two sub-protocols structured roughly as in the Internet MUSE protocol. The first protocol is an unreliable multicast protocol which makes a best-effort attempt to efficiently deliver each message to its destinations. The second protocol is a 2-phase anti-entropy protocol that operates in a series of unsynchronized rounds. During each round, the first phase detects message loss; the second phase corrects such losses and runs only if needed.

Parameters

- zbcast_fanout : the fanout of gossip messages. This determines how many destinations a member gossips to during each round.
- zbcast_sweep: the interval of each round.
- zbcast_idle: how many rounds to wait after the last retransmission request of a message before that message can be garbage collected.
- zbcast_max_polls: the maximum number of destinations a member can poll for missing messages during one round.
- zbcast_max_reqs: the maximum number of retransmission requests a member can make during one round.
- zbcast_max_entropy: the maximum amount of data a member can retransmit during one round.
- zbcast_req_limit: the threshold for message retransmission request before that request is multicasted to the whole group.
- zbcast_reply_limit: the threshold for message retransmission before that retransmission is multicasted to the whole group.

Properties

- Under some conservative assumptions about the network properties, message delivery can be proved to have a bimodal distribution under this protocol: with a very small probability the message will be delivered to a small number of destinations(including failed ones); with very high probability the message will be delivered to almost all destinations; and with vanishingly low probability the message will be delivered to *many* but not *most* destinations.

- Using this protocol for multicast transmissions, virtual synchrony cannot be guaranteed since it admits a non-zero probability of message losses at some operational members. Message losses (if any) are reported to the application. If the message loss is deemed to compromise correct behavior, the application may decide to leave the group and then rejoins them, triggering state transfer – a separate feature provided by Ensemble.
- This protocol needs multicast support from underlying layers. If IP-multicast is not available, GCAST protocol is needed to simulate the effect of multicast by a series of unicasts.
- As its current implementation, this protocol requires *groupd* membership services.
- This protocol assumes that the application is able to control its message transmission within a certain rate (rate-based flow control). If the load injected into the network is heavier than what it can sustain, the failure probability and latency guarantees of the protocol may no longer hold.

Sources

layers/zbcast.ml

Generated Events

Dn(ECast)
Dn(ESend)
Up(ELostMessage)

Testing

- Extensive experiments have been conducted on a SP2 parallel machine (used as a network of UNIX workstations) with group size ranging from 8 to 128 nodes. The protocol scales gracefully and maintains stable throughput.
- The protocol has been tested on a multicast-capable LAN with 30 Solaris workstations. One of the member is the sender and the rest are receivers. The sender is sending at a rate of 200 messages per second. Each message is 1000 bytes. Most of the receivers are able to maintain a steady throughput of 200 msgs/sec. Message losses are very rare.

  We emphasize that in both tests we have reached the limit of the largest group of machines which we have access to. We believe that our protocol can scale far more than what is indicated above.
- In the next step of our work, we will investigate its performance on WAN.

This layer and its documentation were written by Zhen Xiao. It is based on the PBCAST *protocol implemented by Mark Hayden. This documentation is based the* Bimodal Multicast *paper.*

9.35 VSYNC

Virtual synchrony is decomposed into a set of 8 independent protocol layers, listed in figure 3. The layers in this stack are decribed in the layer section.

[TODO: here describe the overall protocol created by composing all the protocol layers]

Parameters

- **[TODO: composition of parameters below]**

Protocol

[TODO: composition of protocols below]

Properties

- **[some form of composition of properties in layers]**

Notes

- Causal ordering can be introduced by replacing the MNAK layer with a causal implementation of same protocol.
- Weak virtual synchrony can be implemented by removing the SYNC layer and adding application support for managing multiple live protocol stacks.

Testing

- Use the various testing code described in the component layers.
- Version of Jan 12, 1996, tested with > 100000 random failure scenarios.
- Version of April 10, 1997, tested with > 100000 random failure scenarios.
- Random testing is done nightly on debugged VSYNC protocol stack.

name	purpose
LEAVE	reliable group leave
INTER	inter-group view management
INTRA	intra-group view management
ELECT	leader election
MERGE	reliable group merge
SYNC	view change synchronization
PT2PT	FIFO, reliable pt2pt
SUSPECT	failure suspcions
STABLE	broadcast stability
MNAK	FIFO, agreed broadcast
BOTTOM	bare-bones communication

Table 3: Virtual synchrony protocol stack

A Appendix: ML Does Not Allow Segmentation Faults

Normally, Ensemble should never experience segmentation faults. When they occur, there are only a few possible causes. We list these below along with fixes. Please inform us if you detect other sources of "unsafety" in Ensemble.

- One of the Ensemble extensions to Objective Caml written in C (in the **socket** directory) may have a bug. Most (all?) extensions have equivalent ML implementations. Use those and see if the bug goes away.

B Ensemble Membership Service TCP Interface

[This is intended as an appendix to the Maestro paper (Maestro: A Group Structuring Tool For Applications With Multiple Quality of Service Requirements). It describes the exact TCP messaging interface to the group membership service described in that paper.]

The description here is of the nuts-and-bolts TCP interface to the **maestro** membership service service described in the Ensemble tutorial. Ensemble also supports a direct interface to this service in ML. Developers using ML should probably use this interface instead. See **appl/maestro/*.mli** for the source code for the interface to this service.

B.1 Locating the service

The membership service uses the environment variable **ENS_GROUPD_PORT** to select a TCP port number to use. Client processes connect to this port in the normal fashion for TCP services. Client processes can join any number of groups over a single connection to a server, so they normally only connect once to the servers.

If you run **groupd** on all the hosts from which your clients will be using the service, then processes can connect to the local port on their host. However, clients are not limited to using local servers, and can connect to any membership server on their system.

If the TCP connection breaks, the membership service will fail the member from all groups that it joined. However, a client can reconnect to the same server and rejoin the groups it was in. If client's membership server crashes, it can reconnect to a different server.

B.2 Communicating with the service

Communication with the service is done through specially formatted messages. We describe the message types and their format here.

messages: Messages in both directions are formatted as follows. Both directions of the TCP streams are broken into variable-length *packets*. A packet has a header of size 8 of which the first 4 bytes are an unsigned integer in network byte order (NBO) giving the length of the message body (not including the header). The next 4 bytes *must* be zero (this is done for internal reasons, which we shall not go into here). The next message follows immediately after the body.

integers: Integers are unsigned and are formatted as 4 bytes in NBO.

strings: Strings have a somewhat more complex format. The first 4 bytes are an integer length (unsigned, NBO). The body of the string immidiately follows the length.

endpoint and group identifiers: These types have the same format as strings. For non-Ensemble applications, the contents can contain whatever the transport service you are using requires. Ensemble only tests the contents of endpoint and group identifiers for equality with other endpoints and groups.

lists: Lists have two parts. The first is an integer giving the number of elements in the list. Immediately following that are the elements in the list, one after the other and adjacent to one-another. It is assumed that the application knows the formats of the items in the list in order to break them up.

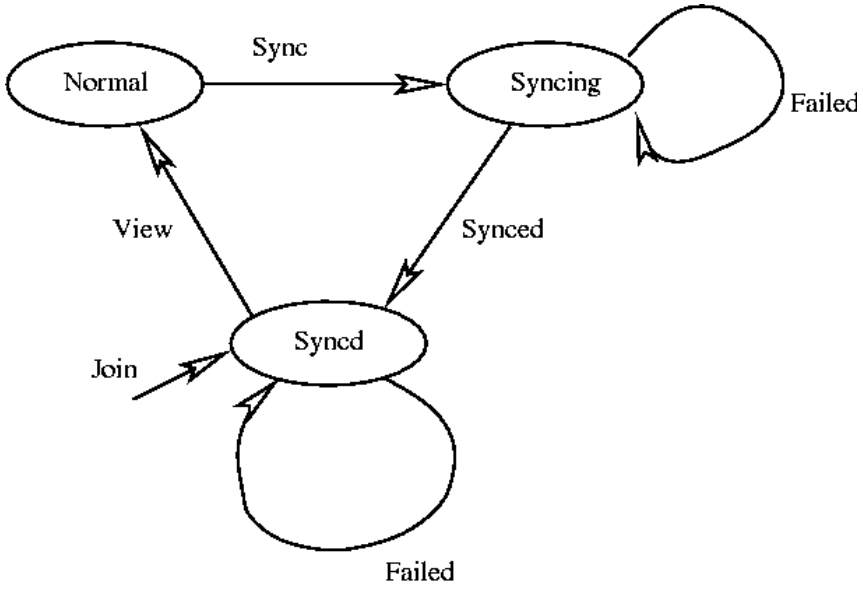

Figure 8: *Client state machine diagram of the client-server membership protocol.*

The actual messages sent between the client and the servers are composed of integers and strings. The first field of a message is an integer *tag* value from which the format of the remainder of the message can be determined.

- **Coord_View** : A new view is being installed. The view is a list of Endpt.id's. A member who just sent a Join message may not be included in the view, in which case it should await the next View message. The ltime is the logical time of the view. The first entry in the view and the ltime uniquely identify the view. The ltime's that a member sees grow monotonicly. In addition, a boolean value is sent specifying whether this view is a primary view. The primary bit is based on the primary bit of the group daemon's being used.

integer	Coord_View = 0
group	my group
endpoint	my endpoint
integer	logical time
boolean	primary view
endpoint list	view of the group

- **Coord_Sync** : All members should "synchronize" (usually this means waiting for messages to stabilize) and then reply with a SyncOk message. The next view will not be sent until all members have replied.

integer	Coord_Sync = 1
group	my group
endpoint	my endpoint

- **Coord_Failed** : Fail a member is being reported as having failed. This is done because members may need to know about failures in order to determine when they are synchronized.

integer	Coord_Failed = 2
group	my group
endpoint	my endpoint
endpoint list	failed endpoints

- **Member_Join** : Request to join the group. Replied with a View message.

integer	Member_Join = 3
group	my group
endpoint	my endpoint
bool	logical time

- **Member_Sync** : This member is synchronized. Is a reply to a Sync message. Will be replied with a View message.

integer	Member_Sync = 4
group	my group
endpoint	my endpoint

- **Member_Fail** : Fail other members in the group (or leave the group by failing self).

integer	Member_Fail = 5
group	my group
endpoint	my endpoint
endpoint list	failed endpoints

- **Client_Version** : This is sent by the client process to tell the server its version. If the server's version is incompatible, the server will send an Error message and close the client's connection. The value to use for the version field can be found by running any of the Ensemble demonstration programs with the **-v** flag.

integer	Member_Version= 6
string	service name ("ENSEMBLE:groupd")
string	my version ("0.40")

- **Server_Error** : This is sent by the server. An error has occurred. Usually this means the client's connection will be closed.

integer	Server_Error = 7
string	explanation

C Bimodal Multicast (by Ken Birman, Mark Hayden, and Zhen Xiao)

There are many methods for making a multicast protocol *reliable*. The majority protocols in Ensemble aim to provide virtually synchronous properties. However, these properties come with a price in terms of the possibility of unstable or unpredictable performance under stress and limited scalability. This is unacceptable to some applications where system stability and scalability are viewed as inextricable from other aspects of reliability.

This section describes a bimodal multicast protocol that not only has much better scalability properties but also provides predictable reliability even under highly perturbed conditions. This work is described in the *Bimodal Multicast* paper (ncstrl.cornell/TR98-1683) by Ken Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu and Yaron Minsky (this documentation is based on that paper). The original version of the protocol was implemented by Mark Hayden. It was reimplemented by Zhen Xiao with many new optimizations and is described in the **ZBCAST** layer. In the remainder of the section, we will refer to our new protocol as Zbcast.

C.1 Protocol description

Zbcast protocol consists of two stages:

- During the first stage, the protocol multicasts each message using an unreliable multicast primitive. IP multicast can be used to serve this purpose if it is available. Otherwise a randomized tree-dissemination protocol is used (the GCAST layer). In the latter case, the protocol uses the Ensemble group membership manager to track membership information.
- The second stage uses an anti-entropy protocol to detect and correct message losses in the group. The protocol consists of a series of rounds. In each round, every member randomly choose another member and exchange its message histories with him. A member compares the received message history with its own. If it detects itself to be lacking a message during the exchange, then it solicits copies of that message from the original sender.

  A member records the round in which a message is received. The message will be gossiped until it has not been requested for retransmission for a prespecified number of rounds. At that point the member will garbage collect that message. Due to the probabilistic nature of the protocol, it is possible for a message to be garbage collected by other members while some operational member has not received it yet. In such cases, the member missing the message will report a message gap to the application.

C.2 Usage

To use Zbcast protocol, specify the "Zbcast" property on the command line as follows(using perf demo as an example):

```
perf -prog 1-n -add_prop Zbcast -groupd
```

This assumes that IP-multicast is available in the underlying network. Remember to set the related environment variables:

```
ENS_DEERING_PORT=38350
ENS_MODES=Deering:UDP
```

Otherwise the Gcast layer needs be linked into the stack:

```
perf -prog 1-n -add_prop Zbcast -add_prop Gcast -groupd
```

Note that in both cases we need *groupd* to track group membership information. This is the state of art of the current implementation and is not something intrinsic to the protocol. If sufficient needs arise, we are going to remove this restriction.

Message losses are reported to the application via **Up(ELostMessage)** event. The application can either ignore those messages (i.e. multimedia applications) or leave the process groups and then rejoin them, triggering state transfer.