Appia Group Communication

Alexandre Pinto
apinto@di.fc.ul.pt

Oct 2005

Contents

1 Introduction

In this document we assume you are familiarized with *Appia* [5] **A**pplication **P**rogram **I**nterface [1].

Despite *Appia* being designed to support several communication paradigms, it is offered with support for *Group Communication*. To achieve this, *Appia* offers several protocols. The protocols were developed to provide all the necessary mechanisms to enforce *virtual synchrony*[2].

Virtual synchrony is a *Group Communication* paradigm that basically states that all participants in a group, see the membership of the group in the form of *views*, and that all see the same *view* changes in the same order. There is a total ordering of the *views*. It also states that all messages sent within a *view* are delivered in that *view*, and if a *view* change is necessary then all messages of the current view are delivered before the new *view* is installed. Therefore, to provide a correct view change three major steps are necessary:

1. The group must be blocked, to guarantee that step 2 will terminate.
2. All group messages received by any member are delivered to all members before the new view.
3. The new view is delivered to all members.

The *view* changes occur because we expect the groups to be dynamic, that is, members will fail and leave the group, while new members will join the group.

Appia Group Communication protocols where inspired heavily on those present in the *Ensemble* [4] system. Many names are common, although some of the functionality was changed to adapt to the different needs and capabilities of *Appia*.

This document presents an introduction to those protocols, it explains how they should be put together and presents an example.

2 Protocols

All the *Group Communication* protocols are necessary only to guarantee the correctness of the view changes. In fact they all work only to that end. The rest of this manual assumes that when the term "messages" is used we mean any event descendent of *GroupSendableEvent*. That class is the superclass of all events that will be sent to other members.

The *Group Communication* layers assume that reliable point-to-point communication with FIFO order is offered by the underlying layers.

2.1 View Change - Intra

The view change is performed by the *intra* protocol. However this protocol doesn't guarantee the correctness of the view change. This is done by the *sync* protocol. The *intra* protocol is implemented by the IntraLayer and corresponding IntraSession. It starts the view change in response to either a failure or a ViewChange request. It then sends a NewView event that will go from top downwards. The protocols that guarantee the correctness of the view change capture this event and only release it when their respective protocols have ended. After reception of the NewView event, the IntraSession will send a PreView event that will also go from top downwards. This event contains a proposal for the new view to install, without any failed members. Layers that wish to change the composition of the view to be installed, change the view in this event. Upon reception of the PreView event, the IntraSession will install exactly the same view it received in this event. This steps are performed only at the coordinator. The propagation of the new view to the rest of the group is done using the InstallView event. All the members retransmit the event when they receive it, ensuring that all members receive the new view before any message sent in that view, without *casual order* being offered by the underlying layers.

2.2 Virtual Synchrony Enforcer - VSync

The correctness of the view change, particularly regarding the *virtual-synchrony* paradigm, is guaranteed by the *sync* protocol and implemented by the VSyncLayer and corresponding VSyncSession. When the VSyncSession receives a *NewView* event it starts the protocol by sending a Block event to all members. After the protocol has started a BlockOk event will go through the channel from top downwards. All layers should accept this event. The reception of the event notifies the layer that the group must be blocked, prior to a view change, and that the layer must stop sending messages until the new view is installed. The layer may hold the event for the time necessary to send any important messages, but when it releases the event it agrees that it will not send any more messages. Failure to do so may cause unpredictable behavior. When a member is blocked, it replies to the view coordinator with information regarding the number of messages it has sent and received from each member. These counters are sufficient due to the reliable FIFO properties offered by the underlying layers. After all members are blocked, the coordinator will send to all members information regarding the messages they must receive in the *Sync* event. The *sync* protocol, relies on the *stable* protocol 2.3 to deliver the same messages to all members. When all members have received all the messages they are supposed to, ie, when the messages received counters reach the desired value, they reply with the *Sync* event, and the coordinator will release the *NewView* event, allowing the view change to continue.

2.3 Reliable Group Message Delivery - Stable

As said, the *sync* protocol relies on the *stable* protocol to deliver all the view messages to all the members. As such the *stable* protocol, implemented by the StableLayer and corresponding StableSession, in effect offers *reliable broadcast*[3]. The protocol uses a table containing the messages received from each member (y axis), for all members (x axis). Each member has a replica of the table. The table is constructed by each member periodically sending its line of the table, ie, the messages received by that member from each member. The messages sent by the member corresponds to the messages received from himself. The information is *piggybacked* in regular group messages. During normal operation the StableSession only maintains the mentioned table, and keeps a copy of each message received from another member, in a StableStorage object. Due to the reliable FIFO properties offered by the underlying layers, it doesn't keep copies of its sent messages. When a member fails the tables are consulted to see if some of the messages sent by the failed member were received by a subset of the remaining members. If so the members that didn't receive these messages, ask the lowest rank member that received the messages to resend them.

2.4 View Merging - Inter and Heal

As with *Ensemble*, in *Appia Group Communication* protocols there isn't a *join* operation, in other words, there isn't a mechanism by which a member may join a group. Instead there is only the merge mechanism, in which two different views of the same group are merged together to form a new view. This mechanism is necessary to withstand network partition and recovery. In practice this means that when a member wishes to join a group, it creates a view with a single member, himself, and then the merge protocol will take over and merge that view with an existing view, if one exists.

There are concurrent views when two or more distinct views of the same group coexist simultaneously. This can be due to network dynamic reconfigurations causing network partitions. This is a problem that is solved by merging the concurrent partitions. The implemented merge mechanism is divided in two parts:

1. Concurrent view detection.
2. Concurrent view merge.

Lets start by the latest. The view merge is performed by the *inter* protocol, implemented by the InterLayer and corresponding InterSession. Currently it allows for several different views to merge in a single view change. It functions by implementing a *flooding consensus*[3] algorithm that will decide on the ordered set of views that will merge. The new view is inserted in the view change process, by capturing the PreView event and replacing the contained view by the new merged view. Communication between the coordinators of the views to merge must be reliable FIFO, therefore the layers below the *Group Communication* are used.

View detection is performed by the *heal* protocol, implemented by the HealLayer and corresponding HealSession. Detection is done using any one of the following mechanisms:

- An external server, the GossipServer;
- A dedicated (*gossip*) *multicast* address;
- The *multicast* messages sent by the views in regular operation, that are received by the other views.

Communication with the GossipServer or through the dedicated *multicast* address is done through a different *Appia* channel, that must offer only FIFO order and message failure detection. Deliver guarantee isn't necessary. The bridge between the main *Group Communication* channel and the gossip channel is done by the GossipOutLayer and corresponding GossipOutSession. The event used in that communication is the GossipOutEvent. Only the view coordinator sends *gossip* messages, that announce the view existence.

The GossipServer is an external service that receives messages from processes and retransmits the messages to all known processes, that is processes from which it has already received messages. It can be viewed as a mean to achieve a *pseudo-broadcast*. To tolerate network partitions the service may function as a group of distributed, replicated, servers.

2.5 Other

The remaining layers are devoted to several auxiliary functions. The failure detection is performed by the *suspect* protocol, implemented by the SuspectLayer and corresponding SuspectSession.

There is a simple protocol named *bottom*, implemented by GroupBottomLayer and corresponding GroupBottomSession. As the name implies, the layer must be the bottommost layer of the *Group Communication* layers. The layer performs the interface between the *Group Communication* layers above, and the point-to-point layers below. For instance it converts from group Endpt to network addresses.

3 Events, Message and ExtendedMessage

Communication between layers in *Appia* is done with *events*. To pass values between layers there are two possible ways. Within the same stack, the values can be passed in the attributes of the event object. Between layers in different stacks, different processes, the values must be passed within the *Message*. The *Message* is a component of all *SendableEvent*s and it constitutes the payload of all messages that are sent through the network. If a layer wishes to add its own header to the message that will be sent to other processes, then it adds the header to the events *Message*.

In the design of the *Message* the main inspiration came from *x-kernel* messages. There were some changes to adapt it to Java, but the functions are exactly the same, although with different arguments. In Java all data structures are objects, therefore the usual header would also be an object. Because Java offers the possibility to *serialize* an object, *Appia* offers a new class that extends the original *Message* with direct object manipulation, pushObject and popObject. This new class is called *ExtendedMessage* and retains all functionality previously offered by *Message*. Due to Java serialization restrictions, all objects that are put in a ExtendedMessage must implement the *java.io.Serializable* interface. The ExtendedMessage also offers methods to insert and retrieve all basic types.

Due to the object serialization mechanisms of Java, pushing and popping an object using ExtendedMessage has performance deficiencies.

4 Session development How-To

A new application session that uses Appia *Group Communication* protocols, must satisfy the following rules:

1. The *layer* must specify that it provides a GroupInit event and accepts View and BlockOk events.

2. The group communication protocols must be started by sending a GroupInit event. It must contain the initial view. To set/get the information necessary to create the initial view, it is normally required that:

   (a) A RegisterSocketEvent is sent and the response received, specifying the *IP port* to which the transport session (UDP or TCP) will bind. The returning RegisterSocketEvent will contain the member address.
   (b) If *IP multicast* is to be used, then a MulticastInitEvent must be sent and the response received, specifying the multicast address to use.

3. The session must handle BlockOk events. When such event is received the layer must eventually forward it (go() method). After that the session must not send new group events until a new view is received. It may still receive events sent by other members.

4. The session must handle View events, that notify the install of a new view.

5. All events sent to the group must descend from GroupSendableEvent. The dest field is always ignored[1]. The source field is only valid in received events, and contains the identification (Endpt) of the sender. The orig field is also only valid in received events and contains the rank in the current view of the sender.

5 Example

With the distribution comes an example application. The application can be executed with the following command:

```
java -cp ¡Appia classes path¿ demo.Appl -port ¡IP port¿ -gossip { ¡IP
host:port of the gossip server¿ — ¡gossip multicast address¿ }
```

The Appl class does only channel initialization, while the application functionality is performed by classes in the “appia.test.appl” package.

As said, a Gossip server may be required. The server may be started using the following command:

```
java -cp ¡Appia classes path¿ gossip.GossipServer [-port ¡IP port¿]
```

The “port” is the IP port where the server will be receiving messages. Defaults to 10000.

6 Protocol Summaries

6.1 bottom

Layer name : GroupBottom

Synopsis : Group Bottommost layer

Description : Interface between *Group Communication* layers and *point-to-point* layers.

Provided events :

[1]See documentation regarding the appia.protocols.group.events.Send on how to send a message to a subset of the view members

- **OtherViews** ():
  Notifies the reception of an event from a different view of the group.

Used events :

- **View** (*require*):
  The current group view.
- **GroupSendableEvent** (*accept*):
  Up events are checked if they belong to the current view. The group and view id is added to **Down** events.
- **GroupEvent** (*accept*):
  Checks if they belong to the current view.
- **GroupInit** (*accept*):
  Gets IP multicast address, if provided.
- **OtherViews** (*accept*):
  Sets the behavior regarding events received from a different view of the group.
- **Debug** (*accept*):
  Starts showing debugging information.

Layer dependability : Must be the lowest of all *Group Communication* layers.

Author : Alexandre Pinto

6.2 suspect

Layer name : Suspect

Synopsis : Suspects possible failures.

Description : *Failure detector.* Propagates suspicions to other group members.

Provided events :

- **Alive** ():
  Periodically sent to all group members, if no application messages are sent, notifying this member is still alive.
- **Suspect** ():
  Used to propagate suspicions to other members.
- **Fail** ():
  Local notification that a member has failed.
- **SuspectTimer** ():
  Timer to send *Alive* messages.
- **EchoEvent** ():
  The *Fail* event is sent upwards within an *echo* event to go through the entire stack.

Used events :

- **View** (*require*):
  The current group view.
- **Alive** (*accept*):
  Periodically received, notifying the sender member is still alive.
- **Suspect** (*accept*):
  Used to receive suspicions from other members.
- **GroupSendableEvent** (*accept*):
  Avoids the sending of *Alive* messages.
- **FIFOUndeliveredEvent** (*accept*):
  Sent by the *FIFO* layer, notifying it couldn't deliver an event. The destination member is suspected.
- **TcpUndeliveredEvent** (*accept*):
  Sent by the *TcpComplete* layer, notifying it lost communication with a member. The destination member is suspected.
- **SuspectTimer** (*accept*):
  Timer to send *Alive* messages.
- **Debug** (*accept*):
  Starts showing debugging information.

Layer dependability :

Author : Alexandre Pinto

6.3 sync

Layer name : VSync

Synopsis : Enforces virtual-synchronous view change.

Description : Blocks all group members, and checks if all members have received the same messages, before the view change.

Provided events :

- **Block** ():
  Used to notify the group members that they should block. Used also to gather information regarding messages received.
- **BlockOk** ():
  Used to notify the above layers that they must stop sending messages.
- **Sync** ():
  Used to inform the group members of the messages they must receive. Used also to notify that a member has received all messages.
- **EchoEvent** ():
  The *BlockOk* is sent upwards within an *echo* event.

Used events :

- **View** (*require*):
  The current group view.

- **NewView** (*require*):
  Received when a view change is necessary.
- **Block** (*accept*):
  Received when the blocking of the group is necessary.
- **BlockOk** (*accept*):
  Received when all the above layers have agreed not to send more messages.
- **Sync** (*accept*):
  Received with information about the messages it must receive.
- **Fail** (*accept*):
  Received when a member fails, that member is excluded from the blocking protocol.
- **Debug** (*accept*):
  Starts showing debugging information.

Layer dependability : Bellow there must be a *Intra* layer.

Author : Alexandre Pinto

6.4 stable

Layer name : Stable

Synopsis : Stabilizes current view messages, offering *reliable broadcast*.

Description : Guarantees that all group messages received by any alive member, are received by all alive members.

Provided events :

- **StableGossip** ():
  Used to disseminate stability information between the group members.
- **Retransmit** ():
  Sent to a specific member to request the retransmission of a set of messages.
- **Retransmission** ():
  The retransmission of a message that requires stabilization. In response to a *Retransmit* event.

Used events :

- **View** (*require*):
  The current group view.
- **StableGossip** (*accept*):
  Stability information from another group member.
- **Retransmit** (*accept*):
  Request to retransmit a set of messages stored.
- **Retransmission** (*accept*):
  The retransmission requested by an earlier *Retransmit* event.

- **GroupSendableEvent** (*accept*):
  Events to stabilize. To **Down** events is added a sequence number. **Up** events are registered has being received and stored for possible retransmissions.
- **Fail** (*accept*):
  Received when a member(s) fails. Tries to stabilize messages from the failed member(s).
- **PeriodicTimer** (*require*):
  StableGossip events are sent periodically, but for optimization it is used a periodic timer set by another layer, for example Suspect.
- **Debug** (*accept*):
  Starts showing debugging information.

Layer dependability :

Author : Alexandre Pinto

6.5 intra

Layer name : Intra

Synopsis : Intra view change.

Description : Performs a view change. It interacts with other layers to guarantee a correct view change.

Provided events :

- **View** ():
  Propagates locally a new group view.
- **InstallView** ():
  Propagates to other members a new group view.
- **NewView** ():
  Notifies the start of a view change.
- **PreView** ():
  Contains a proposal of the composition of the new view.
- **EchoEvent** ():
  The *View* event is sent downwards within a *Echo* event, so that all layers receive the new view. The *NewView* and *PreView* events are sent upwards within a *Echo* event, to notify all above layers.

Used events :

- **GroupInit** (*require*):
  Contains the first view.
- **NewView** (*accept*):
  Received when the group is ready to continue the view change.
- **PreView** (*accept*):
  Received with the new view it must install.

- **InstallView** (*accept*):
  Contains a new view to install, sent by the group coordinator.
- **View** (*accept*):
  The new view.
- **Fail** (*accept*):
  A member has failed, so a new view change is required.
- **ViewChange** (*accept*):
  Request to perform a view change.
- **Debug** (*accept*):
  Starts showing debugging information.

Layer dependability :

Author : Alexandre Pinto

6.6 inter

Layer name : Inter

Synopsis : Inter view change.

Description : Performs the merge of two concurrent views of the same group.

Provided events :

- **ViewChange** ():
  To request a view change to the *Intra* layer.
- **MergeEvent** ():
  Used to communicate with the coordinators of other views to merge.
- **MergeTimer** ():
  Each execution of the merge algorithm has a minimum and maximum duration, implemented using two timers.

Used events :

- **View** (*require*):
  The new view.
- **MergeTimer** (*accept*):
  A merge will only terminate after the reception of the timer corresponding to the minimum duration, and will abort if the timer corresponding to the maximum duration is received before the algorithm ends.
- **PreView** (*accept*):
  Received when the group is ready to change view. Resent when the merge protocol has ended, with the merged view to install.
- **MergeEvent** (*accept*):
  Received from the coordinators of the views to merge.
- **ConcurrentViewEvent** (*accept*):
  Received from the *Heal* layer, notifying there is a concurrent view of the group, and a merge may be required.

- **Debug** (*accept*):
  Starts showing debugging information.

Layer dependability : Below there must be an *Intra* layer. Above there must be a *Heal* layer.

Author : Alexandre Pinto

6.7 heal

Layer name : Heal

Synopsis : Concurrent views detection.

Description : Detects the existence of concurrent views of the group.

Provided events :

- **ConcurrentViewEvent** ():
  To notify the *Inter* layer that a concurrent view exists and a merge is necessary.
- **GossipOutEvent** ():
  Used to communicate with the *GossipServer*.
- **HelloEvent** ():
  Used to communicate with concurrent views through the group multicast address, if used.

Used events :

- **View** (*require*):
  The current view.
- **PeriodicTimer** (*require*):
  The group and view id are disseminated periodically. For optimization it is used a periodic timer set by another layer, for example Suspect.
- **GossipOutEvent** ():
  Event received with a group and view id.
- **Debug** (*accept*):
  Starts showing debugging information.

Layer dependability : Below there must be an *Inter* layer. If multicast is not used then below there must be a *GossipOut* layer.

Author : Alexandre Pinto

6.8 leave

Layer name : Leave

Synopsis : Leave protocol.

Description : Gracefully removes a member from the group. Forces a view change.

Provided events :

- **ViewChange** ():
  To request a view change to the *Intra* layer.
- **ExitEvent** ():
  Used to notify the leaving members they have left the group.

Used events :

- **View** (*require*):
  The new view.
- **PreView** (*require*):
  The new view that will be installed. Released without the members that requested to leave.
- **LeaveEvent** (*accept*):
  Request to leave the group.
- **ExitEvent** (*accept*):
  Received when the member has left the group.
- **Debug** (*accept*):
  Starts showing debugging information.

Layer dependability : Must be above the *Stable* layer. Below there must be a *Intra* layer.

Author : Alexandre Pinto

References

[1] Appia protocol development manual. *http://appia.di.fc.ul.pt/docs/appia-pdm.pdf.*

[2] K. Birman. Virtual synchrony model. Technical report, Cornell University, July 1993.

[3] Rachid Guerraoui and Luis Rodrigues. *Introduction to Reliable Distributed Programming.* Springer, 2005.

[4] M. Hayden. *The Ensemble System.* PhD thesis, Cornell University, Computer Science Department, 1998.

[5] Hugo Miranda, Alexandre Pinto, and Luís Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of The 21st International Conference on Distributed Computing Systems (ICDCS-21)*, page to appear, Phoenix, Arizona, USA, April16–19 2001. IEEE Computer Society.