Appia Protocol Development Manual

version 2.1

Hugo Miranda Alexandre Pinto Nuno Carvalho
Luís Rodrigues

Departamento de Informática

Faculdade de Ciências

Universidade de Lisboa

{hmiranda,apinto,nunomrc,ler}@di.fc.ul.pt

December 2005

Abstract

This document describes the Protocol Programming Interface (PPI) of *Appia* for the construction of protocols. *Appia* is a layered communication framework implemented in Java and providing extended configuration and programming possibilities. The conceptual model behind *Appia* is described in several papers (e.g. [3]).

The PPI is presented in two different ways. The former presents the classes signature and details its usage and function. The later is "function-oriented", grouping functions towards the accomplishment of one specific task.

Contents

Chapter 1

Overview

Networked inter-process communication requires that several distinguishable properties be combined in order to provide the derived service. Some networking standards have been developed and are now widely used. This is the case of the Internet Protocols such as IP, TCP, UDP [7, 8, 6] and the OSI model [9]. Most of them assume a layering model, having each protocol piled over another. Each protocol relies on the documented properties of the protocols below to provide his service to the layers above. Transmission Control Protocol (TCP), for instance, relies on routing capabilities of IP to ensure that the sent packets will be delivered to the correct destination. As IP does not ensure FIFO ordering, TCP provides this property.

Each combination of layers (protocols) on the stack provide a different set of properties[1] and can be considered as the service provided by the stack. The properties resulting from each combination and the protocols used in the stack are used interchangeably in this document and referred as the *Quality of Service* (QoS).

Appia is a layered communication support framework. Its mission is to define a standard interface to be respected by all layers and facilitate communication among them. *Appia* is protocol independent. That is, the framework layers any protocol as long as it respects the predefined interface, making no provisions to validate the final composition result.[2]

1.1 *Appia* concepts

This section briefly describes the concepts and terminology used in *Appia*.

[1] Different ordering of protocols can also provide different set of properties.

[2] In fact, *Appia* provides a limited form of stack validation.

Static and dynamic concepts *Appia* presents a clear distinction between the declaration of something (either a protocol or a stack) and its implementation.

A **Layer** is defined by the properties that a protocol requires and those it will provide. A **Session** is a running instance of a protocol. Sessions are always created on behalf of a layer and its state is independent from other instances.

A **QoS** is a static description of an ordered set of protocols. A **Channel** is a dynamic instantiation of a QoS. Protocol instances (sessions) communicate using a channel infrastructure.

All this concepts are illustrated on Figure 1.1 and Table 1.1.

As they are static, layers do not exchange information between them. Instead, they declare the communication interface of their dynamic instantiations: the sessions. Communication between sessions and with the channel is made using **Events**. *Appia* provides a predefined set of events, each with a different meaning but programmers are encouraged to extend this set to detail protocol specific occurrences. Starting from the session who generated it, events flow through the stack on a predefined direction. The information contained in any event extends a basic set of fields that all events must contain.

Reusability Reusability in *Appia* is based on inheritance. Since most of the protocols depend (at least weakly) on the service provided by others, upgrading some may produce incompatibilities. *Appia* uses inheritance to make the upgrades transparent. When a new version of a protocol is released, it is expected that the generated events will have richer information than the previous version. Assuming that none of the previously provided information format is changed, protocols may simply create new events extending previous ones. This way, protocol backward compatibility is assured.

Optimization Inheritance is also used to improve performance. Timer events, for instance, are generated by protocols (as requests) and handled by the channel. Any session is free to extend the standard timer events, adding information that otherwise would have to be kept in the session state. A reliable delivery layer for instance may include the message to be retransmitted in the timeout request event. When the timeout occurs, the session simply peeks the message from the event and resends it.

Event processing time is reduced by preventing protocol instances from handling unwanted events. Each protocol registers its interest in receiving events of some classes. Instances of classes of events not declared by some

layer are not delivered to the corresponding sessions.

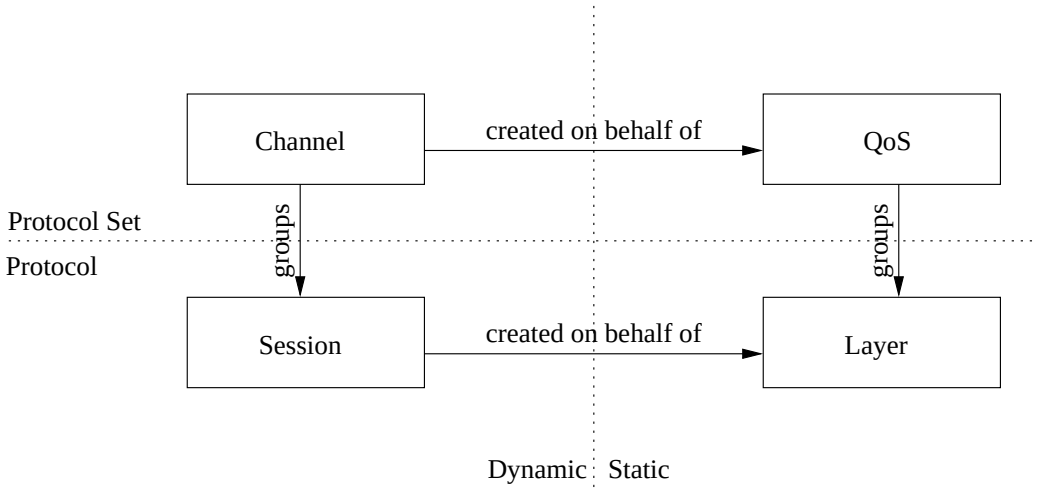

Figure 1.1: Relation between sessions, layers, channels and QoS's

1.2 Protocol definition

Each protocol is defined by two different classes: one extending the basic Layer class and the other extending the Session class. By convention, the former is usually named *Protocol*Layer and the later *Protocol*Session having *Protocol* to be the name of the protocol.

The *Protocol*Layer class is the one participating in QoS definitions. Its purpose is to export the event sets[3] and to create instances of the *Protocol*Session class.

The *Protocol*Session class is the one participating on channels and executing protocol instances. It has two main goals: to cooperate in channel definitions and to handle and generate events, providing the properties expected from the protocol.

1.2.1 Relation between sessions and channels

In *Appia*, a session (i.e. a running instance of a protocol) may participate in several channels simultaneously even if they have different QoS's. This means that a single protocol instance can participate in multiple protocol combinations.

[3] See section 3.1 on page 30.

Concept	Type	#	Description
Layer	Static	1	The static description of a protocol. Declares the properties it requires and provides.
Session	Dynamic	n	Execution instance of a protocol. Keeps the protocol state and implements the properties described in the corresponding layer.
QoS	Static	1	An ordered set of layers. Describes the properties that a running instance of that combination of protocols would have.
Channel	Dynamic	n	An ordered set of sessions, modeled by one QoS. The entity providing the set of properties specified in the QoS.

The expected number of instances per protocol/protocol set in an *Appia* process.

Table 1.1: Relation between static and dynamic concepts in *Appia*.

This is one of the innovative aspects of *Appia* and offers a new perspective in the way different kinds of data are related. For instance, by having only one single FIFO session on two channels, one with an appropriate QoS for video transmission and another for audio, the receiver imposes the sending order of messages across the two media without any additional programming effort.

Whether sessions deal transparently with multiple channels or not is implementation and protocol dependent. On event reception, sessions are free to query the event's channel. Events can be forwarded without sessions knowing the channel being used.

1.3 Implementation classes

There are eleven classes relevant for protocol implementation in *Appia*: QoS, Channel, ChannelCursor, Layer, Session, Event, Message, MsgWalk, MsgBuffer, Direction and Appia. Appendix A presents the UML model of the framework. Remaining classes of *Appia* are not presented as they do not provide relevant features to protocol development.

1.4 Notation

In the rest of the document, methods and classes are presented using usual object-oriented languages notation. Method's classes are always prefixed of the class name. A dot is used as the separator between the class name and the method name.

For example, the line

```
Channel QoS.createUnboundChannel(String channelID)
```

presents method createUnboundChannel from class QoS. The method takes a String as the input and returns a reference to a Channel object.

Classes always have an upper-case first character while methods are identified by a lower-case first character. The remaining characters will be lower cases except when a new word is started.

The existence of optional arguments is signaled by the presentation, like different methods with the same name, of all possible combinations of arguments.

1.5 Changes from previous versions

1.5.1 From version 2.0

Changed the description

Introduced the notion of priority on events. Changed the description of Message and ExtendedMessage classes. ExtendedMessage became deprecated. Changed the API of memory management.

The list of modified interfaces, together with pointers for the sections where they can be found are summarized on table 1.2.

1.5.2 From version 1.2

Introduced Section 3.2.1 that briefly describes the usage of XML to configure *Appia* channels. The class Timer has now a different semantics. Introduced class ExtendedMessage and TimeProvider. Updated UML diagrams on Appendix A.

The *Appia* PPI suffered several changes since version 1.2. The list of modified interfaces, together with pointers for the sections where they can be found are summarized on table 1.3. The name of this document was also updated.

Change	Description	Sections
Class Event	Events now are introduced in the event scheduler with some priority, that is defined in the event.	2.7
Message and ExtendedMessage	The methods of the class ExtendedMessage were moved to the Message class and ExtendedMessage became deprecated.	2.16, 2.13
Memory Management	Changed the MemoryManager API in order to have different thresholds for UP and DOWN events.	2.17, 3.4.8

Table 1.2: Summary of differences from version 2.0

1.5.3 From version 1.1

Section 2 now presents a more detailed description of those classes relevant for protocol implementation and channel definition.

The *Appia* PPI has suffered minor and focused changes since version 1.1. The list of modified interfaces, together with pointers for the sections where they can be found are summarized on table 1.4.

1.5.4 From version 1.0

This document has only suffered minor changes from version 1.0. The list of modified interfaces, together with pointers for the sections where they can be found are summarized on table 1.5.

Change	Description	Sections
Class Direction	All methods of the class Direction are now deprecated. This class is used now just for constants definition. The methods that used this class where modified to receive an integer.	2.6, 2.7, 2.12
Class EventQualifier	All methods of the class EventQualifier are now deprecated. This class is used now just for constants definition. The methods that used this class where modified to receive an integer.	2.8, 2.12
Class Timer	Timers are created with a different semantics. In the previous versions of the Timer class, the Timer was created with an absolute time; now is created with a relative time.	2.11
Class ExtendedMessage	Introduced class ExtendedMessage. This class provides a more clean API to push, pop and peek basic types.	2.16
Memory Management	Introduced the description of memory management in *Appia* channels.	2.17, 3.4.8
Class TimeProvider	Introduced class TimeProvider, used by Sessions to get the system time.	2.18, 2.2
XML	Introduced the API to configure *Appia* channels using a XML configuration.	3.2.1

Table 1.3: Summary of differences from version 1.2

Change	Description	Sections
Insertion of asynchronous events in channels	The procedures for inserting asynchronous events on channels was changed. The previous model can be used (is tagged as deprecated) but is no longer referred in this document. The Async event type was removed from the model.	2.2, 2.7, 3.4.2
Enriched ChannelCursor interface	Methods for jumping directly to a position are now available	2.3
Enriched SendableEvent interface	A new set of constructors was added to the SendableEvent class	2.12

Table 1.4: Summary of differences from version 1.1

Change	Description	Sections
Simplified ChannelCursor interface	Provides a more intuitive, easy to use interface.	2.3, 3.2
Removed constructor from Message class	The same functionality is now provided by a method.	2.13, 3.4.5

Table 1.5: Summary of differences from version 1.0

Chapter 2

PPI description by class

Protocols are implemented by the refinement of three classes: Layer, Session and Event. Layer and Session detail the desired protocols behavior and Event its message passing requirements. Programmers should be aware that layers and sessions are tightly coupled as the former presents the static behavior of a protocol and the later its dynamic one.

2.1 Class QoS

A Quality of Service is a set of properties, each independently provided by one protocol.

QoS mission is to glue protocols (presented as layers), partially validate the resulting composition and define the interaction rules between the protocols. At QoS definition time, layers declare the events they are interested to receive. Using this knowledge, QoS builds for each event class an "event path", including only the layers that are interested in receiving it. The information extracted can then be used to create more efficient channels.

Class QoS defines the Qualities of Services that will be available to the application. From the programmers point of view, a QoS instance is simply an array of layers.

One optional argument of the createUnboundChannel method is the EventScheduler. *Appia* configuration options allow programmers to define event scheduling policies by redefining this class. The default implementation of the EventScheduler class is single threaded and puts all events in a FIFO queue. The internals of the EventScheduler class lie outside the scope of this document and can be found elsewhere [2].

```
class QoS {

    QoS(String qosID, Layer[] layers) throws AppiaInvalidQoS;
    Channel createUnboundChannel(String channelID,
        EventScheduler eventScheduler);
    Channel createUnboundChannel(String channelID,
        EventScheduler eventScheduler, MemoryManager mm);
    Channel createUnboundChannel(String channelID);
    Channel createUnboundChannel(String channelID,
        MemoryManager mm);
    Layer[] getLayers();
    String getQoSID();
}
```

2.2 Class Channel

Channels are instantiations of QoS's. Channels glue sessions the same way QoS's glue layers. A Channel is created on behalf of a QoS type. When a channel is created, it inherits the knowledge captured from the layers in the corresponding QoS, improving performance. On channel creation, event paths are exported from the QoS. The channel maps the layers on the QoS event paths into the binded session to route events.

Channels also provide the background run-time environment for session execution. They are responsible, for instance, for providing timers. The ChannelEvent sub-class of events is dedicated to these operations.

Channel definition Upon creation, a channel is as an array of "typed empty slots". Each of these slots must be filled with a session of the layer specified in the QoS for that position. Sessions can be bound to the slots explicitly (by the user) or implicitly by other sessions (automatic binding). New sessions will be bound by default to the remaining slots not explicitly or implicitly bounded.

Using explicit binding it is possible to associate specific sessions to specific channels. These sessions may either be already in use by other channels or may be intentionally created for the new channel. Explicit binding enables the user to have fine control over the channel configuration.

Using automatic binding it is possible to delegate on already bound sessions the task of specifying the remaining sessions for the channel. A mixture of explicit and automatic binding can be used, with the actions taken by the former having precedence over these taken by the later.

Both explicit and automatic binding are performed over a ChannelCursor object, requested to the Channel. Explicit binding must be performed prior to calling the start method of the channel. One of the tasks of this method is to ensure that every slot is fulfilled with a valid position. The first step performed by start is to invite sessions explicitly bounded to perform automatic binding by calling their boundSessions method. For those slots not explicitly or automatically bounded, the start method requests to the corresponding layer the creation of a new session.

Channel initialization and termination A channel is instructed to start and stop by its methods start and end. Besides the operations concerning session instantiation performed by start, both methods introduce an event in the channel. The ChannelInit event is supposed to be the first to flow in a channel. Protocols should be aware that events created in response to handling a ChannelInit event must be inserted after invoking the go method on the ChannelInit event. Although this requirement is not mandatory and does not produce inconsistencies to *Appia*, other protocols may rely on this property.

The end method introduces a ChannelClose event in the channel. Sessions receiving the ChannelClose event may not introduce more events in the channel but must be prepared to receive others. Received events may be propagated.

A stopped channel may latter be restarted by calling again the start method. However, for temporary suspensions, protocols should consider to use EchoEvents to obtain the same behavior.

```
class Channel {

   String getChannelID();
   QoS getQoS();
   boolean equalQoS(Channel channel);
   void start();
   void end();
   ChannelCursor getCursor();
   TimeProvider getTimeProvider();
   boolean isStarted();
   void setMemoryManager(MemoryManager mm);
   MemoryManager getMemoryManager();
}
```

2.3 Class ChannelCursor

Channel cursors are helpers to session bounding in channels. The class provides methods for iteration over the channel stack, retrieve references to already defined sessions and set sessions for the empty slots. Methods of this class raise AppiaCursorException exceptions to signal operations.

Initially, the cursor is not positioned over the channel. The initial position must be defined by either the top or bottom methods. Scrolling below the bottommost position of the channel or above the uppermost will also raise the AppiaCursorException with an indication of the error occurred.

```
class ChannelCursor {

   void top();
   void bottom();
   void jumpTo(int position) throws AppiaCursorException;
   void down() throws AppiaCursorException;
   void up() throws AppiaCursorException;
   void jump(int offset) throws AppiaCursorException;
   boolean isPositioned();
   Session getSession() throws AppiaCursorException;
   void setSession(Session session) throws AppiaCursorException;
   Layer getLayer() throws AppiaCursorException;
}
```

2.4 Class Layer

Layers are the static representative of micro-protocols. They describe the behavior of micro-protocols. Layers are used on QoS definition to reserve a specific position for a session implementing the protocol and to declare the needed, accepted and generated events, respectively on the evRequire, evAccept and evProvide attributes.

Layers are responsible for instantiating sessions (in response to calls to method createSession) and are notified by the channel whenever one session is dismissed by a channel (by calls to the method channelDispose).

```
class Layer {

   Class[] getProvidedEvents();
   Class[] getRequiredEvents();
   Class[] getAcceptedEvents();
   Session createSession();
   void channelDispose(Session session, Channel channel);
}
```

2.5 Class Session

A session is the dynamic part of a micro-protocol. Sessions maintain state of a micro-protocol instance and provide the code necessary for its execution. Channels provide the connection between the different sessions of a stack. A session keeps a relation of "one-to-many" with channels: one single session can be part of multiple channels. A session is defined as *channel-aware* if its algorithm recognizes and acts differently upon reception of events flowing from different channels. Many of the protocols that can be found in existing stacks are channel-unaware. When a channel is being defined, sessions already bound to the channel may be invited to bound other sessions. The invitation is made by a call to the boundSessions method.

Sessions communicate with their environment by events. Reception of events is made on the handle method. A session can learn the channel that is delivering an event to it by querying the channel attribute of the Event.

```
class Session {

   Layer getLayer();
   void boundSessions(Channel channel);
   void handle(Event event);
}
```

2.6 Class Direction

Class Direction is an implementation support class of *Appia* events. It defines values for the direction it is flowing. The direction attribute of events accepts two values Direction.UP and Direction.DOWN defined as static constants on class Direction.

```
class Direction {
   int direction;
   static final int UP=1;
   static final int DOWN=-1;
   static int invert(int direction);
}
```

2.7 Class Event

Sessions use events to communicate with other sessions and the *Appia* kernel. This class contains the attributes necessary for the event routing. In *Appia*, events can be freely defined by the protocol programmers as along as all inherit from the main Event class. Programmers should be aware that sub-classing should be done as deep as possible on the sub-classing tree, improving event sharing and compatibility among different micro-protocols.

The Event class has three attributes that must be defined prior to the event insertion in the channel. For each, a pair of set and get methods is defined. The attributes are:

direction Stating the direction of the movement (up or down).

channel Stating the channel where the event will flow.

source Stating the session that generated the event. This attribute is important to determine the event route.

The attributes can be defined either by the constructor or by the individual *set* methods. When methods are used, the method init must be invoked after all attributes are defined and prior to the invocation of the go method.

The cloneEvent method uses the Java clone method of the Object class. Redefinitions of this method should always start by invoking the same method on the parent classes.

Since version 2.1, *Appia* events have also an optional attribute – the priority. The default value of this attribute is `DEFAULT_PRIORITY`, but this value can be set between `MIN_PRIORITY` and `MAX_PRIORITY`. The priority of the event is used to upon insertion of the event in the Channel and defines its priority in the event queue.

2.7.1 Concurrency control

Appia is not thread-safe in the sense that consistency is not ensured if protocols insert events in the channel while not owning the *Appia* main thread. However, a thread-safe event method, with a particular semantic, is provided.

The asyncGo method should be called only when an event is inserted asynchronously (i.e. concurrently with the *Appia* main thread) in the channel. If the direction defined at the event is UP, asyncGo will place the event at the bottom of the channel. Otherwise, the event will be placed at the top of the channel. The event will then present the same behavior as any other, respecting the FIFO order while crossing the channel and only visiting the sessions of the protocols that declared it in the accepted set. Events inserted in a channel using the asyncGo method should not be initialized either by the constructor or by the init method.

Asynchronous events are particularly useful for protocols using their own thread, like those receiving information from outside the channel. Examples of such protocols are those listening to a socket to retrieve incoming messages. When an incoming network message arrives, the session can use these events to request the delivery of the *Appia* main thread or to insert incoming messages in the channel.

Note: Protocol programmers should be aware that the asynchronous insertion of events in the channel must be handled with particular care has it subverts the event usual behavior. Events inserted asynchronously initiate their route at one of the ends of the channel. This does not respect possible causal dependencies between events. Furthermore, programmers should be aware that the use of asynchronous events may subvert the ordering of the stack. Consider the example of the previous paragraph. If some protocol is below the protocol receiving messages from the network, it should not be presented with incoming network messages, that are expected to be sent toward the top of the stack. This problem is most likely to occur if the event type used for the asynchronous event is the one used for sending the message to the stack.

```
class Event {

   static final int MINPRIORITY=0;
   static final int DEFAULTPRIORITY=127;
   static final int MAXPRIORITY=255;
   Event(Channel channel,int dir,Session source) throws
        AppiaEventException;
   Event();
   void init() throws AppiaEventException;
   void setDir(int direction);
   int getDir();
   void setChannel(Channel channel);
   Channel getChannel();
   void setSource(Session source);
   Session getSource();
   void setPriority(int priority) throws InvalidParameterException;
   int getPriority();
   void go() throws AppiaEventException;
   void asyncGo(Channel c, int d) throws AppiaEventException;
   Event cloneEvent() throws CloneNotSupportedException;
}
```

2.8 Class EventQualifier

The event qualifier class differentiates channel events with one of three values: ON, OFF and NOTIFY. The precise interpretation of this values will depend on the qualified event type. However, a common usage pattern is defined:

ON is used for setting requests or starting a mode or operation. OFF is intended for abortion of requests or mode cancellation. NOTIFY is used for notifications of occurrences.

```
class EventQualifier {

   static final int ON=0;
   static final int OFF=1;
   static final int NOTIFY=2;
}
```

2.9 Class ChannelEvent

The ChannelEvent class is the topmost class grouping all channel related events. That is, all events provided by the channel or containing requests of services provided by the channel. This class inherits from the main Event class and includes the attribute qualifier of type EventQualifier, allowing to determine the type of operation to be performed. Instances of the ChannelEvent class are never created. Its subclasses are used to detail the requested or provided operation.

```
class ChannelEvent extends Event {

  void setQualifierMode(int qualifier);
  int getQualifierMode();
}
```

2.10 Class EchoEvent

EchoEvent events are event carriers. When a EchoEvent reaches one of the sides of the channel, the event passed to the constructor is extracted and inserted in the channel in the opposite direction. No copies are realized: the inserted object instance is the same that was given to the EchoEvent.

EchoEvents allow protocols to, for example, perform composition introspection, like learning the available maximum PDU size, or perform requests to other protocols like temporarily suspending the channel activity.

The carried event will be initialized by *Appia* prior to being inserted in the channel. The main Event class attributes will be set as if the event has been launched by the channel. The protocol launching this event must declare himself as the provider of the event.

```
class EchoEvent extends ChannelEvent {

  EchoEvent(Event event, Channel channel, int dir,
       Session source);
  Event getEvent();
}
```

2.11 Classes Timer and PeriodicTimer

Appia offers periodic and aperiodic timer notification services. The direction the event flows and the EventQualifier attribute of the event distinguish requests from notifications. Table 2.1 presents the expected combinations. The

attributes declared by a Timer extend those available in the ChannelEvent with a String and the time (after the current time), in milliseconds, that the notification should occur. When issuing a timer request, the eventQualifier attribute must be set to ON.

Operation	Direction	Qualifier
Request	DOWN	ON
Cancellation	DOWN	OFF
Notification	UP	NOTIFY

Table 2.1: Expected combinations of Directions and Qualifiers on Timers operations in an *Appia* execution

Programmers are encouraged to extend the basic Timer class. This will impact performance at two different levels. If the event type declared on the provided and accepted events for the protocol matches the newly defined event type, notifications requested by other protocols will not consume wasteful resources on this protocol. On the other hand, the new class may encompass any information required by the protocol to handle the timeout. This improves protocol execution time. When the timeout is delivered to the protocol, it delivers the same object instance. The qualifier attribute will be set to NOTIFY and the direction attribute will have a value inverse to the one defined at timer request.

Cancellation of a timer is requested by creating a new timer event with the same timer ID and a OFF qualifier. Note that event cancellation can not be ensured by *Appia*: the notification event may already be inserted in the channel when the cancellation reaches the bottom of the channel.

```
class Timer extends ChannelEvent {

  Timer(long interval, String timerID, Channel channel,
      int direction, Session source, int qualifier) throws
      AppiaException;
  void setTimeout(long period);
  long getTimeout();
}
```

The semantic associated with PeriodicTimer events is that a notification is due every "period" milliseconds. *Appia* only ensures that no more events than periods expired will be raised.

The object delivered upon timer expiration will be a copy of the original object. The copy is performed using the cloneEvent method. Specialization

can also be used to redefine this method in order to provide a different semantic from that initially defined which is to perform a deep copy of all attributes except the timerID (which has its reference copied). If redefined, cloneEvent should start by calling its parent cloneEvent method. After issuing a request to cancel a periodic timer, an undefined number of notifications, those already inserted in the channel, can be received.

```
class PeriodicTimer extends ChannelEvent {

  PeriodicTimer(String timerID, long period, Channel
       channel, int direction, Session source, int qualifier)
       throws AppiaException;
  void setPeriod(long period);
  Time getPeriod();
}
```

Note: *Appia* provides weak time delivery guarantees for notification as this may compromise the event FIFO ordering within the channel. The only provided guarantee is that notifications will be raised by the timer manager *after* the requested timeout period has expired.

2.12 Class SendableEvent

SendableEvents are one of the brunches of the event tree defined by *Appia*. The semantic expected to be applied by protocols regarding SendableEvents is that those are the events to be sent to the network. Non SendableEvents are supposed to be local to the channel that created them.

SendableEvents extend the basic event class with three attributes: source, dest and message. Due to their strong dependence on protocols, network and implementation language, the former are of type java.lang.Object. Their instantiation type is supposed to be agreed by the protocols composing the channel and can even change while the event crosses the stack. It is expected that most of the protocols use them transparently relying only in equality operations. It is therefore advised that value based comparison operations should be defined for the chosen class.

Appia it self does not provide any support sending or delivering events to/from the network. This task must be done by some protocol that interfaces a channel with a socket.[1] When retrieving SendableEvents (or any of its subclasses) from the network, protocols are expected to satisfy at least the

[1] *Appia* distributions already provide some protocols with these functionalities.

following conditions on the event inserted in the channel of the receiving endpoint:

- All attributes of the Event class should be correctly filled; The source session of the event is the session that retrieved the event from the network and will insert it in the channel;
- The values of the source and dest attributes are equal to the ones received by the session that sent the event to the network;
- The message attribute has the same sequence of bytes received by the session that sent the event to the network;
- The event type should be the same;

Note that besides the event type, no special requirements are imposed for sending subclasses of SendableEvents. In particular, attributes not inherited from SendableEvent are not expected to be passed to the remote endpoint. This is the behavior of the current protocols that interface the network, namely UDPSimple, TCPSimple and TCPComplete.

Messages are set and retrieved by two specific operators. Class Message is defined in section 2.13.

```
class SendableEvent extends Event {

  Object dest;
  Object source;
  SendableEvent(Channel channel, int direction, Session
       source) throws AppiaEventException;
  SendableEvent(Message msg);
  SendableEvent(Channel channel, int direction, Session
       source, Message msg) throws
       AppiaEventException;
  Message getMessage();
  void setMessage(Message m);
}
```

2.13 Class Message

The class Message abstracts an array of bytes with methods providing efficient operations for adding and removing headers and trailers. The class was

conceived as the principal method for inter-channel communication.[2] Message provides an interface for sessions to push and pop headers of byte arrays. Message interface is mainly imported from the *x*-Kernel [5]. The use of message was devised assuming that the layer responsible for sending messages to the network has weak serialization capabilities.

The class has only an empty constructor. To initialize a message instance with an array of bytes, one should call setByteArray, specifying the first position in the source array and the number of bytes to be copied. Other methods take a MsgBuffer as an argument.[3] All push, peek and pop operations (which respectively add, query and extract an header) are called with the len attribute of MsgBuffer defined. The remaining values are ignored and overlapped by the method execution. When the call returns, the off attribute points to the first position in the data buffer where the header is stored or can be retrieved.

The sequence of actions performed to push an header is:

1. Prepare a MsgBuffer object with the length of the header;
2. Invoke the push method;
3. Copy the header to the data array, starting at the position indicated by offset;[4]

Popping an header requires the same sequence of actions to be performed, retrieving the data in step 3.

For usability reasons, this class that was extended with methods that provide automatic insertion and removal of all basic types. These methods use the basic methods to push, pop and peek data and can provide the same performance results, with a cleaner protocol code.

Note: The byte array presented to the protocol will typically be larger than the required length. Most of the times, the remaining positions will have headers of other protocols in the channel. *Appia* takes no provisions to ensure that protocols respect their self defined boundaries.

[2] *Inter-channel* communication is defined as the mean by which channels on different processes exchange information. This is the opposite of *intra-channel* communication, ideally performed by event attributes.

[3] The goal of the MsgBuffer is to avoid memory copies. This class is described later in this document.

[4] The only restriction is that the header must be defined prior to calling the go method on the event owning the message, so, to avoid memory copies, the header can be constructed directly in the buffer.

Iterating over an entire message (for checksumming or encryption) is made with MsgWalk class.

```
class Message {

    Message();
    void setByteArray(byte[] data, int offset, int length);
    int length();
    void peek(MsgBuffer mbuf);
    void discard(int length);
    void discardAll();
    void push(MsgBuffer mbuf);
    void pop(MsgBuffer mbuf);
    void truncate(int newLength);
    void frag(Message m,int length);
    void join(Message m);
    MsgWalk getMsgWalk();
    byte[] toByteArray(); public void pushObject(Object obj);
    public void pushLong(long l);
    public void pushInt(int i);
    public void pushShort(short s);
    public void pushBoolean(boolean b);
    public void pushDouble(double d);
    public void pushFloat(float f);
    public void pushUnsignedInt(long ui);
    public void pushUnsignedShort(int us);
    public void pushByte(byte b);
    public void pushUnsignedByte(int ub);
    public void pushString(String str);
    public Object popObject();
    public long popLong();
    public int popInt();
    public short popShort();
    public boolean popBoolean();
    public double popDouble();
    public float popFloat();
    public long popUnsignedInt();
    public int popUnsignedShort();
    public byte popByte();
    public int popUnsignedByte();
    public String popString();
```

```
        public Object peekObject();
        public long peekLong();
        public int peekInt();
        public short peekShort();
        public boolean peekBoolean();
        public double peekDouble();
        public float peekFloat();
        public long peekUnsignedInt();
        public int peekUnsignedShort();
        public byte peekByte();
        public int peekUnsignedByte();
        public String peekString();
        public Object clone() throws CloneNotSupportedException;
}
```

2.14 Class MsgBuffer

The MsgBuffer class is used as an helper class for operations over messages. The goal of this class is to improve performance by avoiding message copies.

The MsgBuffer class is used to pass arguments to and receive arguments from methods of the Message class. The fields are used with the following meaning:

data An array of bytes retrieved or to be included in the message;

off The first position in the array data containing information relevant for the operation. Respecting usual array representation, the first position of an array has offset 0;

len The number of bytes of the array data relevant for the operation;

Array data positions not between **off** and **off+len-1** are reserved and can not be used.

Instances of this class have always the same usage pattern: user fills the len attribute of one instance and invokes the method passing the instance as the argument. When the method returns, the data, off and len attributes will be appropriately filled. In peek, pop and next (from the MsgWalk class) the array contains the data retrieved from the message. In push the array contains the space to be filled with the headers by the session.

```
class MsgBuffer {

   byte[] data;
   int off;
   int len;
   MsgBuffer();
   MsgBuffer(byte[] data, int off, int len);
}
```

2.15 Class MsgWalk

MsgWalk objects are iterators over messages. This class is intended to be used by protocols operating over the entire message buffer such has checksum or cipher protocols. The array returned by the next method can be used for reading and writing but no data can be appended or deleted from the message. If there is no more bytes in the message, the call to the next method of this class will put the data attribute to null.

```
class MsgWalk {

   void next(MsgBuffer mbuf)
}
```

2.16 Class ExtendedMessage

This class is **deprecated**. The methods provided by this class were moved to the Message class.

```
class ExtendedMessage extends Message {


}
```

2.17 Class MemoryManager

Memory managers limit the available memory for messages in a channel.

When a header is pushed into a message, the amount of bytes requested is bound to the memory manager. When a Channel receives a SendableEvent, the Message's size is also bounded to the corresponding memory manager. Exceeding the memory manager's available memory raises the runtime exception AppiaOutOfMemory.

When a header is popped from a message, the corresponding amount of bytes is unbound from the memory manager. The amount of bytes used by a Message in a SendableEvent are unbounded from the memory manager when a SendableEvent leaves a Channel.

Headers not popped from a message during its life time are unbounded from the memory manager when the garbage collector cleans the Message object.

```
class MemoryManager {

   MemoryManager(String id, int size, int upThreshold, int
        downThreashold);
   String getMemoryManagerID();
   boolean aboveThreshold(int direction);
   int getThreshold(int direction);
   void setThreshold(int newThreashold, int direction); void
        setMaxSize(int newSize) throws
        AppiaWrongSizeException;
   int getMaxSize();
   int used();
}
```

2.17.1 Using the memory manager

Memory management is enabled by default in the *Appia* distributions, but can be desabled if the user don't need it. Disabling this functionality will improve the *Appia* performance. To deactivate it, edit the appia/AppiaConfig.java file and set the quotaOn boolean to false, then recompile the *Appia* kernel and the protocols that will be used.

A memory manager is assigned by the programmer to one or more channels at channel definition time. Note that the available memory must be shared by every channel using the same memory manager. Class Channel is defined in section 2.13 and Class Message is defined in section 2.2.

2.18 Class TimeProvider

Protocols that need to read the system current time should use the interface TimeProvider. An instance of the default implementation of this interface is obtained by invoking the getTimeProvider method from the class Channel (see section 2.13). This interface provides the system current time in milliseconds or microseconds, as described below:

```
interface TimeProvider {

    public long currentTimeMillis();
    public long currentTimeMicros();
}
```

Chapter 3

PPI description by subject

This chapter details the steps needed to perform:

1. QoS definition
2. Channel creation
3. Channel disposal
4. Event creation and handling

The same methods were presented, grouped by classes in Chapter 2.

3.1 QoS definition

The concept of QoS in Appia is defined in Section 1.1. QoS's are defined by a name and an ordered enumeration of layers:

```
QoS.QoS(String qosID, Layer[] layers) throws AppiaInvalidQoS
```

In order to partially validate the newly formed QoS and improve the performance of the channels created from it, layers export three event related methods:

```
Class[] Layer.getProvidedEvents()
```

where each layer states the events it will generate,

```
Class[] Layer.getRequiredEvents()
```

having the events each layer requires to provided the expected service, and

```
Class[] Layer.getAcceptedEvents()
```

for layers to state the events they are willing to receive. For sanity it is expected the required events to be a subset of the accepted ones.

The default implementations of this methods return the contents of the attributes evAccept, evRequire and evProvide which are also Class arrays.

Users can get instances of Class objects with the static Java API methods:

```
Class.forName(String className)
```

or

```
className.class
```

The QoS constructor will throw an AppiaInvalidQoS exception when at least one event type belongs to any "required event" set but is not member of any "provided event" set.[1] In practice, this exception identifies a particular case of invalid stacks: those containing protocols expecting services not provided by the others. However, the nonexistence of an exception can not be interpreted as a proof of correction: neither event direction nor semantical interpretation are verified on incompatibility analysis.

3.2 Channel definition

Channels are dynamic entities, composed of ordered sets of sessions. Sessions, in turn, create, receive, handle and forward events. All channels are created on behalf of a QoS. This is made by invoking one of the methods:

```
Channel QoS.createUnboundChannel(String channelID)
Channel QoS.createUnboundChannel(String channelID, EventScheduler
    eventScheduler)
Channel QoS.createUnboundChannel(String channelID, MemoryManager
    mm)
Channel QoS.createUnboundChannel(String channelID, EventScheduler
    eventScheduler, MemoryManager mm)
```

[1] Channel generated events are implicitly provided.

The channelID uniquely identifies the channel in the system and should be equal across different endpoints as it is used by protocols receiving events from the network, to learn the destination channel of the event.

The eventScheduler argument allow programmers to specify an event scheduling policy. A predefined event scheduler will be used by default.

A channel is a stack similar to the one of the QoS having layer positions filled by corresponding session instances. Prior to usage, all positions of a channel must be filled (binded) by a session. The sessions filling each slot must have been created by requests to the layer occupying the corresponding position in the QoS. Figure 3.1 relates this concepts in a channel definition.

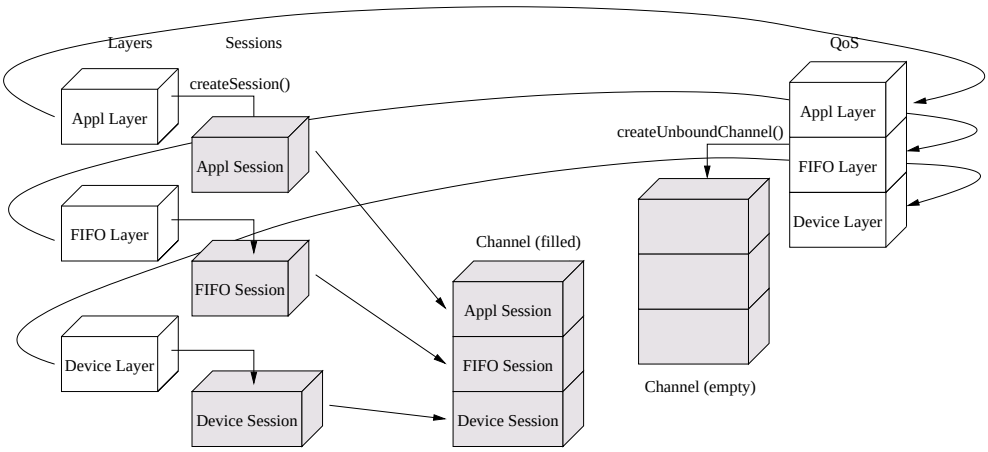

Figure 3.1: Reliable point-to-point channel definition.

The binding of sessions to the corresponding layers is made, for each slot, by the first of the following steps:

Explicitly having the user to explicitly bind a session to a certain position,

By peer sessions Sessions already binded to the channel may explicitly bind others,

By default Positions not binded by any of the above, will be filled by new sessions.

New session instances are created by requests to the appropriate Layer:

Session Layer.createSession()

Browsing through slots is done by a "Channel Cursor", requested to the appropriate channel with:

```
ChannelCursor Channel.getCursor()
```

After its creation, channel cursors should be positioned by one of the following operations:

```
void ChannelCursor.top()
void ChannelCursor.bottom()
void ChannelCursor.jumpTo(int position) throws AppiaCursorException
```

Two methods are provided for channel iteration:

```
void ChannelCursor.down() throws AppiaCursorException
void ChannelCursor.up() throws AppiaCursorException
```

and one for jumping a relative offset:

```
void ChannelCursor.jump(int offset) throws AppiaCursorException
```

offset takes a positive value for moving the cursor *offset* positions up the stack and a negative value for moving the cursor down.

Finally, the method

```
boolean ChannelCursor.isPositioned()
```

signals the user if the cursor was not previously positioned or if a down, up, jump or jumpTo method calls have take the cursor out of the stack.

The method

```
Session ChannelCursor.getSession() throws AppiaCursorException
```

return the current session or a NULL value if the slot is empty.

Binding of a session to the cursor current position is made with:

```
void ChannelCursor.setSession(Session session) throws AppiaCursorException
```

The layer whose instance is expected to fill the slot is obtained with

```
Layer ChannelCursor.getLayer() throws AppiaCursorException
```

AppiaCursorException is a class containing a public attribute type and a set of constants, detailing the occurrence. The values can be:

CURSORNOTSET An operation was attempted prior to the invocation of methods top, bottom, jump or jumpTo,

CURSORONBOTTOM An invalid operation was performed when the cursor was positioned below the lowest position of the stack,

CURSORONTOP An invalid operation was performed when the cursor was positioned above the uppermost position of the stack,

ALREADYSET An attempt to bind a session to an already occupied slot was detected,

WRONGLAYER An attempt to set a session of a layer different of the specified in the QoS for that position was detected.

To allow run-time validation, sessions will have to export its corresponding layer and channels their QoS. The methods for these operations are:

```
Layer Session.getLayer()
```

and

```
QoS Channel.getQoS()
```

The method start of class Channel concludes the steps performed by the user for channel initialization.

```
void Channel.start() throws AppiaDuplicatedSessionsInChannel
```

To allow "By peer sessions binding", those sessions already bound will now be invited to fill the remaining empty slots. Starting from the topmost, every session will be invoked with

```
void Session.boundSessions(Channel channel)
```

it is up to sessions to decide whether some of the empty slots should or should not be filled. In order to fill an empty slot, sessions should obtain a ChannelCursor and perform the operations presented above. Some other methods might also be valuable for this step:

Layer[] QoS.getLayers()
String QoS.getQoSID()
boolean Channel.equalQoS(Channel channel)
String Channel.getChannelID()

Note: The algorithm traverses the channel slots in a strict vertical way: if a session is bound prior to its position be traversed by the algorithm it will also be invited to contribute to it.

After the algorithm has visited all the binded sessions, remaining slots will be automatically binded to new sessions of the matching layer. Finally, a ChannelInit event will travel the channel in ascending direction.

3.2.1 Channel definition using XML

The process of *Appia* channel creation and initialization can be automated using an *Extensible Markup Language* (XML) description. Using XML, the programmer can describe templates for channels, channel instantiations and sharing policies of the sessions that compose the channels. It can also initialize Sessions with properties defined in the XML description.

The following example shows how to build an *Appia* channel with TCP as the bottom layer and an *Ecco* protocol[2] as the top layer.

```
<template name="ecco_t">
  <session name="tcp_s" sharing="global">
    <protocol>appia.protocols.tcpcomplete.TcpCompleteLayer</protocol>
  </session>
  <session name="ecco_s" sharing="private">
    <protocol>appia.test.xml.ecco.EccoLayer</protocol>
  </session>
</template>
<channel name="ecco_c" template="ecco_t" initialized="yes">
  <chsession name="ecco_s">
    <parameter name="localport">4000</parameter>
    <parameter name="remotehost">localhost</parameter>
    <parameter name="remoteport">4001</parameter>
  </chsession>
</channel>
```

Listing 3.1: XML file.

To load a XML configuration and start *Appia* using the previously loaded configuration, the programmer can use the following static methods:

[2]This is an example protocol available on the *Appia* distribution since version 2.0.

```
AppiaXML.load(xmlfile);
Appia.run();
```

More details on the usage of XML to configure *Appia* channels are described in the *AppiaXML* Tutorial [4].

3.3 Channel disposal

Channel disposal is started by invoking:

```
void Channel.end()
```

The closing of a channel is signaled to sessions and layers.

Sessions are notified by the introduction of a ChannelClose event at the top of the channel. Sessions should perform any necessary clean-up procedures upon reception of it (including sending messages) and forward the event only when they are ready to be terminated.

Layers will be notified to allow appropriate garbage collection in implementation languages where it is not automatic. Whether layers perform it or not is layer implementation dependent and will require some kind of interaction between Session and Layer implementation. When the closing event is returned to the channel,[3] *Appia* signals all layers which have generated sessions in the channel with

```
void Layer.channelDispose(Session session, Channel channel)
```

Note that the receipt of a ChannelClose event on a session or a channelDispose call on a layer does not declare that a session is no longer in use as it may still belong to other opened channels.

3.4 Event flow

Event flow is decomposed in two major components: creation and visit to sessions.

An important attribute of an event is the direction it will flow. Class Direction is defined simply as two static constant values (UP and DOWN).

[3] After it has been processed by every session.

An event is created by invoking its constructor. A "do-it-all" constructor is defined at the main Event class:

```
Event.Event(Channel channel, int direction, Session source) throws
        AppiaEventException
```

In this constructor, channel is where the event is intended to flow and source is a reference to the session creating it.

If this constructor is not used, the above arguments must be introduced by separated calls:

```
Event.Event()
void Event.setDir(int direction)
void Event.setSource(Session source)
void Event.setChannel(Channel channel)
```

The individual setting of the above attributes is concluded by

```
void Event.init() throws AppiaEventException
```

An event is not inserted in the channel in any of the above ways. In order to make it flow through sessions, method

```
void Event.go() throws AppiaEventException
```

should be invoked. This is also the method to be invoked by sessions who finish processing it and want it to continue its route through the channel. Events are delivered to sessions by calls to

```
void Session.handle(Event event)
```

The main Event class attributes are queried by

```
int Event.getDir()
Channel Event.getChannel()
Session Event.getSource()
```

The getSource method will return null if the event was generated by the channel.

The AppiaEventException groups all the event related exceptions. Like AppiaCursorException it has a type attribute and a set of constants:

NOTINITIALIZED The event was not properly initialized. It was created using the empty constructor but the init method was not called before go invocation.

ATTRIBUTEMISSING The event was created using the empty constructor and one of the fundamental attributes was not defined prior to calling init.

UNKNOWNQUALIFIER The event qualifier was not properly defined (see Section 2.8).

UNKNOWNSESSION The session that created the event does not belong to the specified channel.

UNWANTEDEVENT The event will not be consumed by any class in the channel.[4]

CLOSEDCHANNEL Attempt to send an event in a session who has previously received a ChannelClose event for the channel specified.

Event sub-classing is a crucial factor in *Appia*. Users are free to extend event classes. Sub-classing improves performance and reusability. The former by avoiding sessions to process events they are not interested in. The latter by allowing old protocols to interact with new versions of those it depends on.

The following sections provide an overview of the *Appia* predefined event subclasses.

3.4.1 Ordering of events

The default event scheduler ensures "First In First Out" (FIFO) ordering of events. It considers that events enter the queue each time the go method is called on the event and leaves it each time the event is delivered to a session by invoking the handle method. The FIFO ordering is also maintained across events that are not presented to some sessions.

For example, consider two events E_1 and E_2 moving toward the top of the channel. E_1 was introduced on the channel prior to E_2. E_1 is scheduled to visit a session S_1 but E_2 is not. All events generated and sent to the channel by S_1 while on the handle call of E_1 will be presented to upper sessions prior to E_2.

[4]The absence of this exception does not prove the contrary because the direction of the events is not taken in consideration.

Users are free to extend the default EventScheduler class, implementing a different behavior. However, they should notice that the default behavior is the one expected by most of the protocols and that changing it may produce unpredictable results.

3.4.2 Multi-thread handling

Appia current implementation uses a single thread model. Protocols are free to implement their own threads as long as there is no interference with the *Appia* core.

The system provides one exception to this model: the Event's method asyncGo has synchronization features and can be called outside the *Appia* thread.

```
void Event.asyncGo(Channel c, int d) throws AppiaEventException
```

asyncGo should be called when a thread running concurrently with the *Appia*'s thread wants to insert an event in a channel. This method should be called only once, when the event is to be inserted. Events inserted with asyncGo need not to be initialized. The event will be inserted at the top of the stack if its direction is DOWN or at the bottom of the stack otherwise. After being inserted, the event presents a behavior that is similar to that of any other event.

3.4.3 Echo Events

Sometimes a protocol will benefit from receiving events it has generated. This is important whenever a protocol can provide optimizations based on information collected from other protocols.[5] When an EchoEvent reaches one of the sides of the channel, the channel extracts the event it contains and sends it, in the opposite direction. The carried event can be of any type.

The constructor for EchoEvent is:

```
EchoEvent(Event event, Channel channel, int direction, Session source)
```

All arguments apply to the EchoEvent and not to the event being carried. Note that the carried event will be initialized by the channel with appropriate arguments: the source will be NULL to indicate this is a channel generated event; the direction will be the opposite to the one the echo event was using

[5] A fragmentation protocol for instance should know exactly the maximum Protocol Data Unit (PDU) size.

and the channel attribute will be copied. Remaining event attributes will not be changed.

3.4.4 Sendable events

Although not mandatory, "sendable events" are those expected to be sent to the network. This class extends the basic Event class with three attributes: source, dest and message.

The source and the dest are two attributes of type Object. The implementation type is deferred to the layer responsible for their delivery and reception from the network and can change while the event is traversing the stack.

3.4.5 Messages

The message attribute of SendableEvent has type Message. The Message interface is designed for sessions to push and pop headers as events flow on them. The Message interface was mainly imported from the similar work realized at the x-Kernel [1, 5].

Messages are constructed by the method:

```
Message()
```

A message can be initialized with an array of *length* bytes starting at *offset* by:

```
void Message.setByteArray(byte[] data, int offset, int length)
```

The SendableEvent's message is obtained by

```
Message SendableEvent.getMessage()
```

and set by

```
void SendableEvent.setMessage(Message m)
```

Current message length is obtained by:

```
int Message.length()
```

Message interface exports the following manipulation operations:

void Message.truncate(int newLength)

Truncates existing message to the first *newLength* bytes. Truncate is used to strip trailers from a message.

void Message.push(MsgBuffer mbuf)

Allocates a buffer of *mbuf.len* bytes at the beginning of the message and returns a byte array and an offset into the array for the beginning of the reserved space. Typically this operation is used for sessions to append their headers.

void Message.pop(MsgBuffer mbuf)

Returns a byte array and a offset to a contiguous buffer of *mbuf.len* bytes that contains the data previously at the front of the message and removes it.

void Message.peek(MsgBuffer mbuf)

Like pop but the message remains unchanged.

void Message.discard(int length)

Like pop but without returning the header.

void Message.frag(Message m, int length)

Removes all but the first *length* bytes from message and assigns them to message m. Message fragmentation can also be done using event copy constructors followed by pop and truncate operations but this operation is faster.

void Message.join(Message m)

Appends the content of message m to the invoked message.

Message iteration The following set of methods allow sessions to iterate over the entire message enabling full message operations such as encryption or checksumming. The center concept is the MsgWalk class, who keeps the information context necessary.

A MsgWalk instance for a message is obtained by:

MsgWalk Message.getMsgWalk()

The message bytes will be returned in bunches by invoking

void MsgWalk.next(MsgBuffer mbuf)

The size of each bunch can not be predicted prior to method invocation and depends on the existing message structure. The returned array can be used for data retrieval and change.

next signals end of message by returning a 0 value on the length argument and the null value in the data array reference. Data appended to the end of the buffer while traversing it with MsgWalk is also returned. Data pushed on the beginning of message after invocation of getMsgWalk will not be retrieved with next.

3.4.6 Channel startup and shutdown

These operations are signaled by two events: ChannelInit and ChannelClose, both descendent of the ChannelEvent class.

The ChannelInit event is the first to flow on a channel. Only one of this events is expected to flow on each channel. The event has UP Direction.

The ChannelClose event is the last one to flow on a channel. After forwarding this event, sessions can no longer send any event on the channel but may still receive messages until this event reaches the bottom of the stack. The event flows DOWN.

3.4.7 Timers

Appia expects two types of timers: periodic and aperiodic. One important attribute of timers is the EventQualifier, inherited from their parent class ChannelEvent.

Event Qualifiers allow event instances reuse. They qualify events with one of three public class constants values:

ON The event performs a request.

OFF The event cancels a request.

NOTIFY The previously requested event has happened.

When a session requests a timer (periodic or not), it creates a corresponding event instance and sets the event qualifier to ON and the direction to DOWN.

Upon timeout, the TimerManager peeks the timer event and changes its source, direction and qualifier attributes. The first to null (all channel generated events share this value), the second to UP and the later to NOTIFY. This way, sessions receive the same event they forwarded. This is strictly true in the case of aperiodic events. Due to its nature, periodic events can not be reused. A clone is used instead. *Appia* uses Event's method cloneEvent for this.

```
Event Event.cloneEvent()
```

cloneEvent can be redefined by sub-classing. Depending on implementation, it can make a shallow copy, a deep copy or a mix of both. All cloneEvent redefinitions should start by invoking its parent class method.

The *modus operandi* of timers was developed having efficiency in mind: upon reception of a timeout notification, sessions need to retrieve information related with the timeout.[6] If the necessary information is passed with the timeout request (by extending the Timer event), it will return with the "happened" event, avoiding the delay of searching it.

Timer and PeriodicTimer classes have similar constructors. Only the second argument differs:

```
Timer.Timer(long period, String timerID, Channel channel, int direction,
        Session source, int qualifier)
PeriodicTimer.PeriodicTimer(String timerID, long period, Channel channel,
        int direction, Session source, int qualifier)
```

3.4.8 Memory management

Memory managers limit the memory available for messages. Exceeding that value raises the runtime exception AppiaOutOfMemory in the corresponding push operation.

A memory manager is created using the following method:

[6] For instance, in a reliable protocol, missing an Acknowledgment will result in a timeout. In order to re-send the message, the session has to find it among all the unacknowledged ones.

MemoryManager.MemoryManager(String id, int size, int upThreshold, int downThreashold)

id is the memory manager identification and size is the maximum value (in bytes) that messages in channels bound to that memory manager can hold. The up and down threshold arguments should be defined between 0 and size and is used to verify if a channel that is using a memory manager, is reaching its maximum capacity, in each flow direction.

String MemoryManager.getMemoryManagerID()

Gets the memory manager identification.

boolean MemoryManager.aboveThreshold(int direction)

This method verifies if the currently used amount of memory reached the specified threshold for the given direction.

int MemoryManager.getThreshold(int direction)

Gets the current threshold (int bytes) for the given direction.

void MemoryManager.setMaxSize(int newSize) throws AppiaWrongSizeException

This method changes the maximum amount of bytes to newSize. If newSize is lower than 0 or lower than the amount currently used by messages, a AppiaWrongSizeException is thown.

int MemoryManager.getMaxSize()

Gets the maximum size in bytes available for this memory manager.

int MemoryManager.used()

Returns the current amount of bytes used.

A memory manager is bounded to a Channel at channel definition time, using one of the methods

Channel QoS.createUnboundChannel(String channelID, EventScheduler eventScheduler, MemoryManager mm)
Channel QoS.createUnboundChannel(String channelID, MemoryManager mm)

Notes:

- Due to a bug in previous Java JRE versions, the memory manager only work on Java JRE 1.3.1 or higher.

3.4.9 Debugging

A particular kind of event type is the Debug class. This class descends from ChannelEvent, inheriting the EventQualifier attribute.

The constructor for Debug events is:

Debug.Debug(Channel channel, int direction, Session source, OutputStream output)

To get the destination of their debugging information, sessions should invoke the method

OutputStream Debug.getOutput()

In debugging context, EventQualifier constants have the following meaning:

ON Session should switch to debugging mode

OFF Session should end debugging mode

NOTIFY Sessions should dump their current state

The state of an event can be obtained by invoking the method

void Event.debug(OutputStream output)

Appendix A

Appia Universal Model Language Diagrams

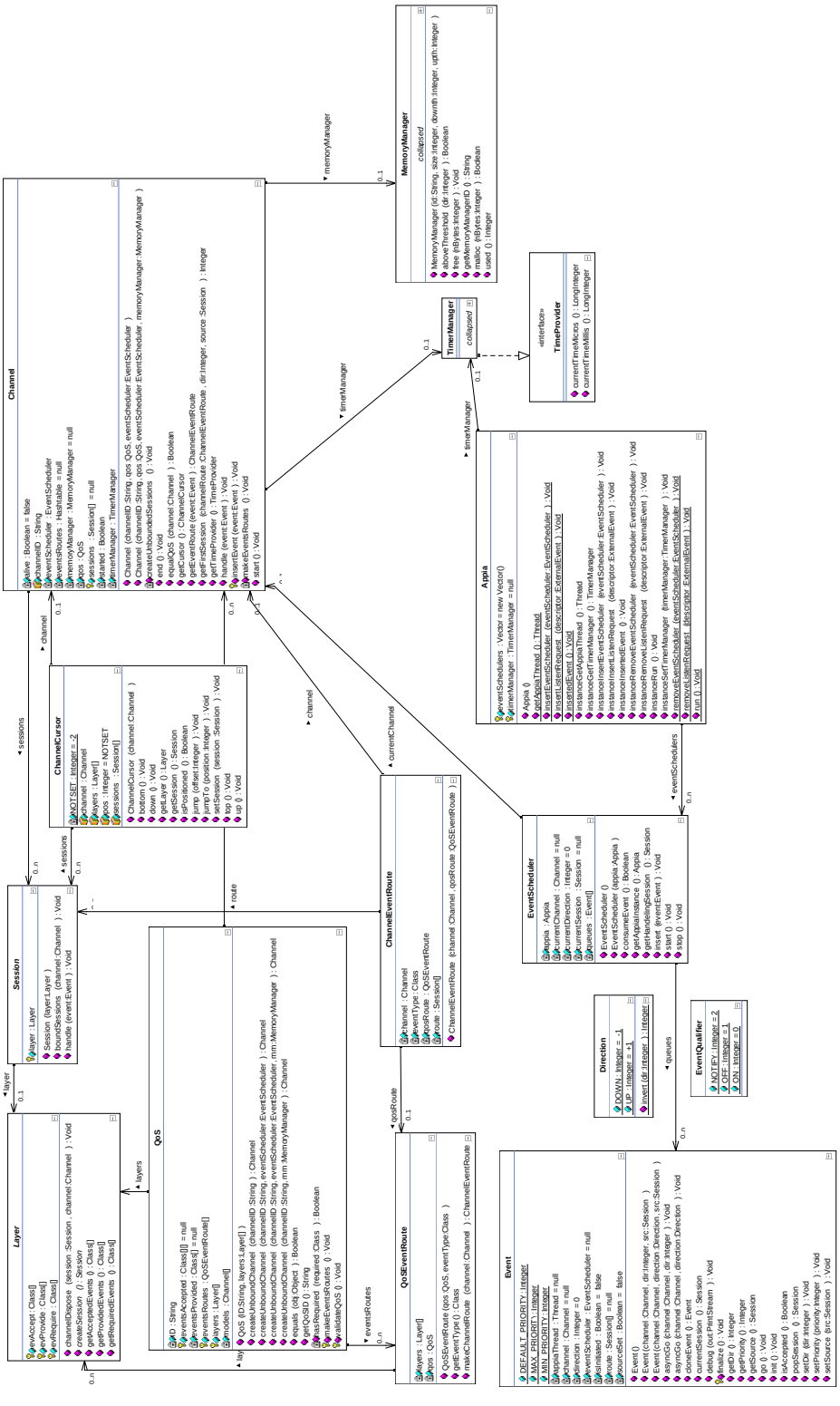

Figure A.1: *Appia* main UML

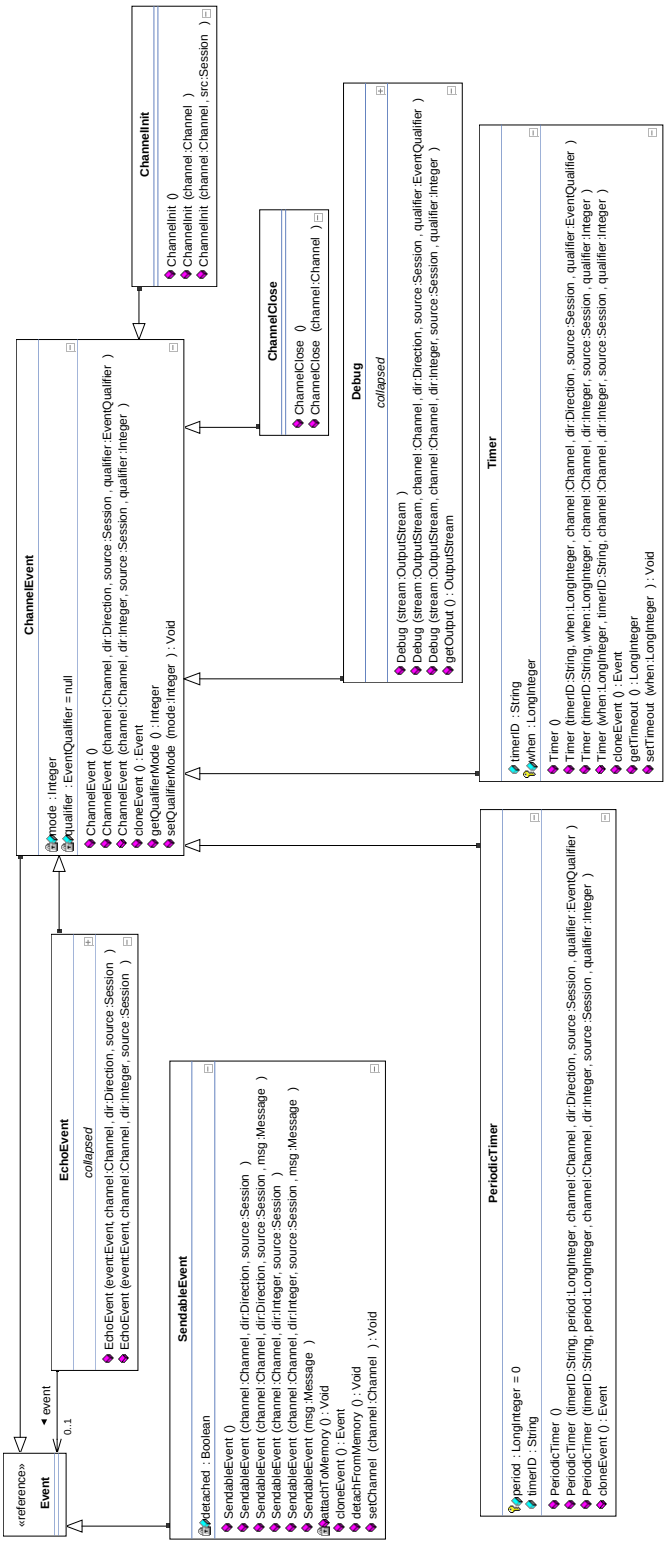

Figure A.2: *Appia* predefined events UML

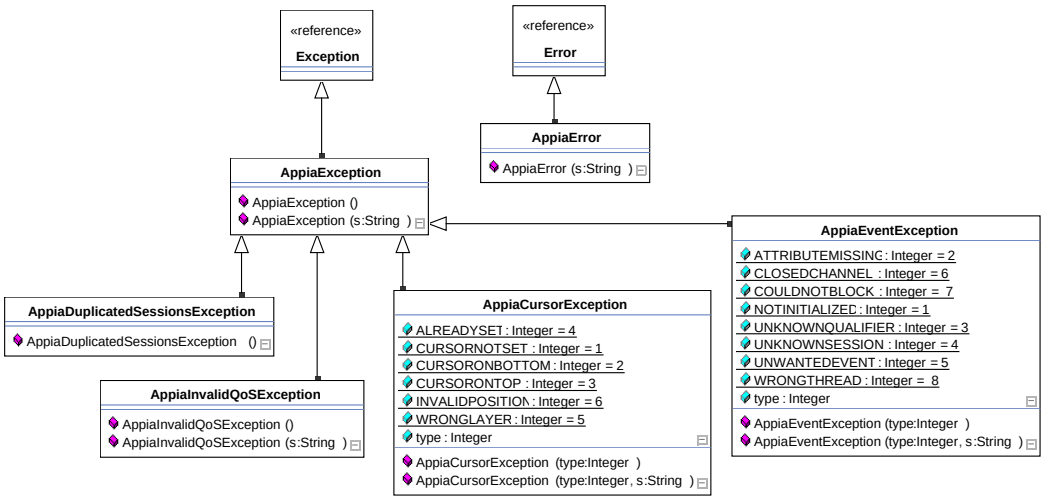

Figure A.3: *Appia* framework exceptions UML

Index

Bibliography

[1] N. Hutchinson and L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[2] Hugo Miranda. Plataforma de suporte ao desenvolvimento e composição de malhas de protocolos. Master's thesis, Departamento de Informática - Universidade de Lisboa, May 2001.

[3] 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)*, pages 707–710, Phoenix, Arizona, USA, April 2001. IEEE Computer Society.

[4] José Mocito, Liliana Rosa, Nuno Almeida, and Luís Rodrigues. *AppiaXML: A Brief Tutorial.* Faculdade de Ciências da Universidade de Lisboa, sep 2004. http://appia.di.fc.ul.pt/documentation.html.

[5] Network Systems Research Group. *x-kernel Programmer's Manual (Version 3.3)*, June 1997.

[6] J. Postel. User Datagram Protocol. Request for Comments 768, USc Inf. S. Inst., August 1980.

[7] J. Postel. Internet Protocol. Request for Comments 791, USc Inf. S. Inst., September 1981.

[8] J. Postel. Transmission Control Protocol. Request for Comments 793, USc Inf. S. Inst., September 1981.

[9] H. Zimmermann. OSI Reference model - The ISO Model of Architectur for Open Systems Interconnection. *IEEE Transactions on Communications*, COM-28(4):425–432, April 1980.

Acknowledgments

Thanks to Bruno Simões, Daniel Barradas, Fernando Vicente, João Martins, José Mocito, Liliana Rosa, Maria João Monteiro, Paulo Sousa, Pedro Vicente, Sandra Teixeira, Sérgio Formigo and Susana Guedes for their valuable help in the development of this project.