Implementing a Protocol Spec.

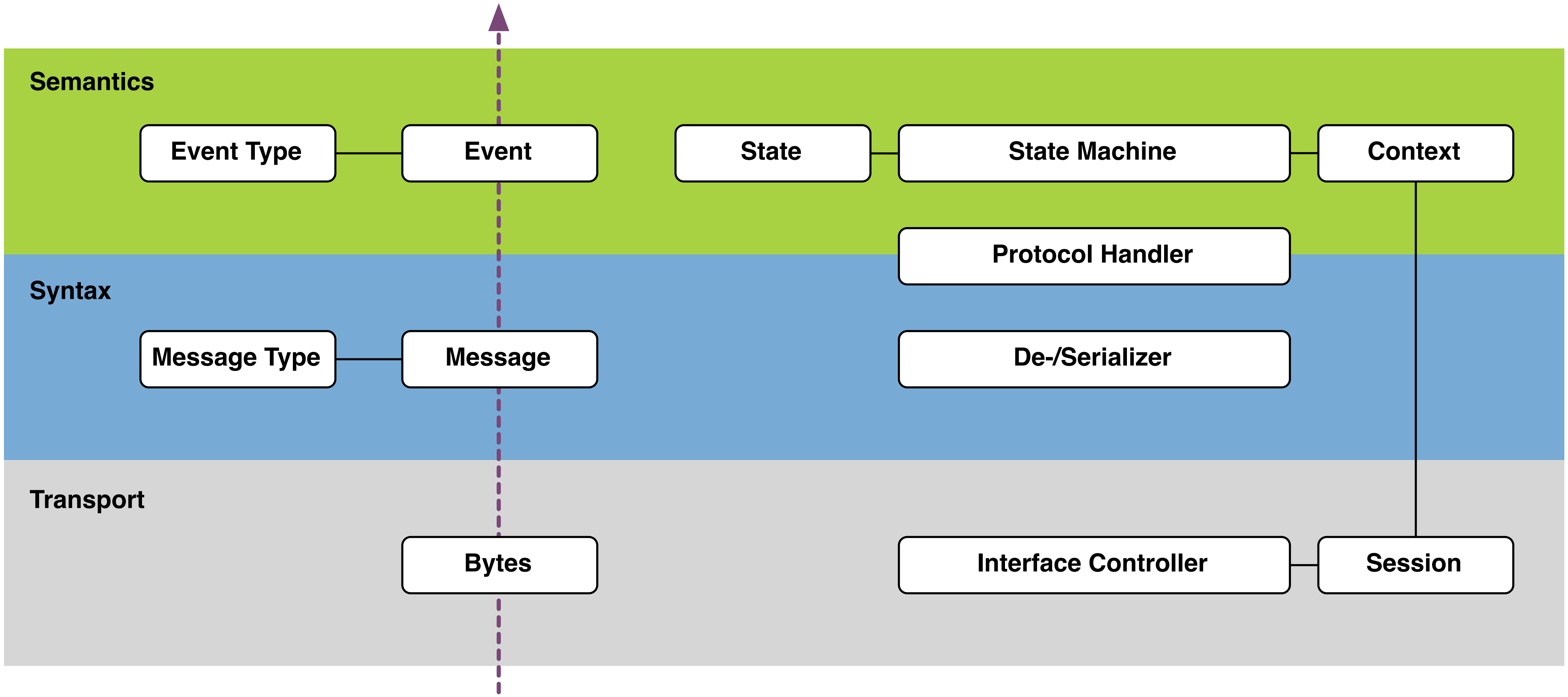
RES, Lecture 5

Olivier Liechti

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

How do I implement my protocol?

The Application Protocol Toolkit


Semantics
Event Type
Event
State
State Machine
Context
Protocol Handler
Syntax
Message Type
Message
De-/Serializer
Transport
Bytes
Interface Controller
Session

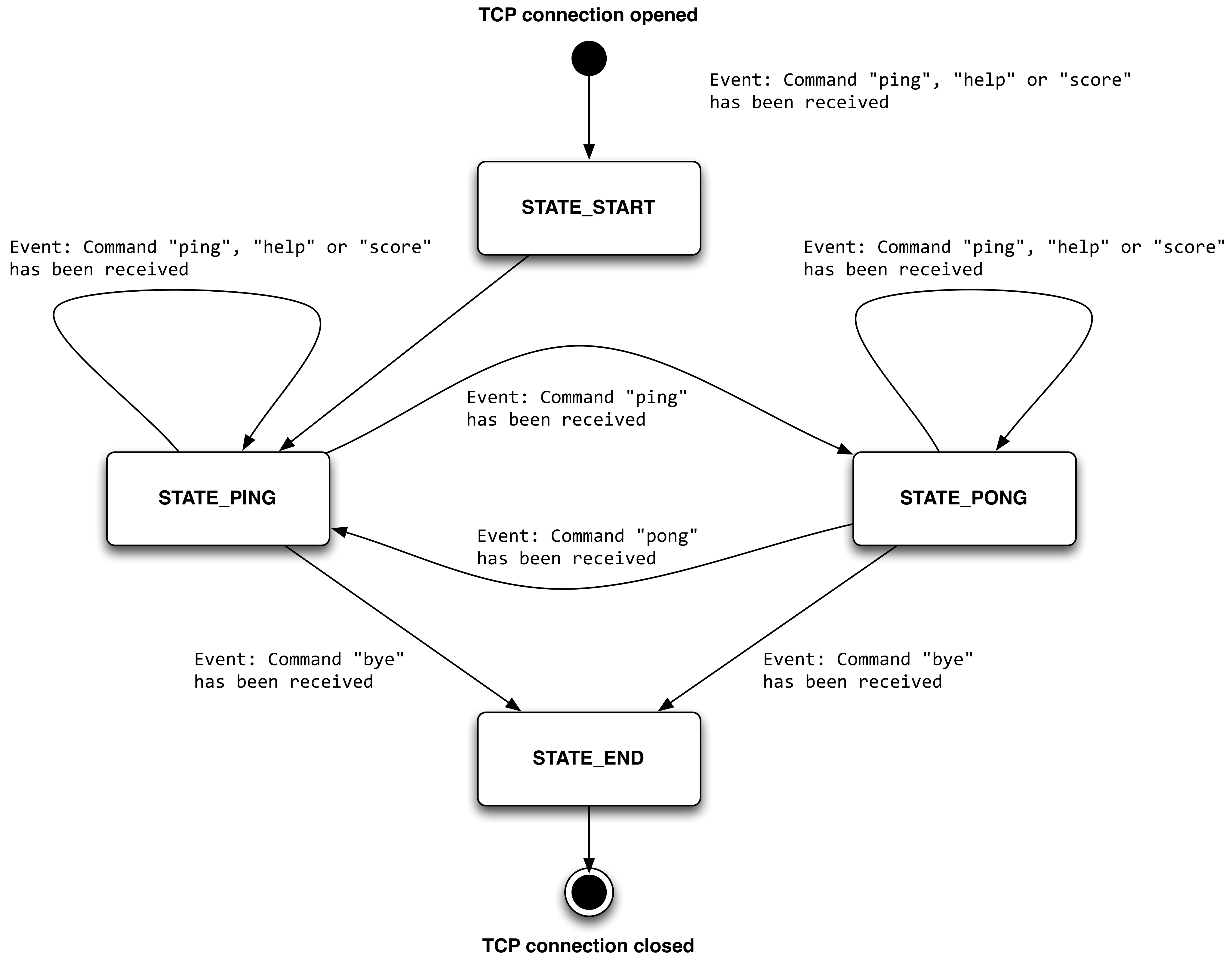
```java
public interface IInterfaceController {

     /**
      * This method is used to start the interface controller
      */
     public void startup();

     /**
      * This method is used to connect a protocol handler with the interface
      * controller. As a result, when data arrives on the interface, the controller
      * will ask the protocol handler to provide a message deserializer. Once it
      * has obtained a protocol message from the raw data, it will pass it to
      * the protocol handler
      * @param handler the protocol handler
      */
     public void registerProcotolHandler(IProtocolHandler handler);

     /**
      * This method is used to get the registered protocol handler
      * @return the protocol handler connected with the interface controller
      */
     public IProtocolHandler getProtocolHandler();

     /**
      * This method is used to send a protocol message to a client, in the context
      * of a particular session.
      * @param sessionId the id of the session to which the message belongs
      * @param m the message to send
      */
     public void sendMessage(long sessionId, Message m);

     /**
      * This method is used to close a session
      * @param sessionId the id of the session to close
      */
     public void closeSession(long sessionId);
}
```

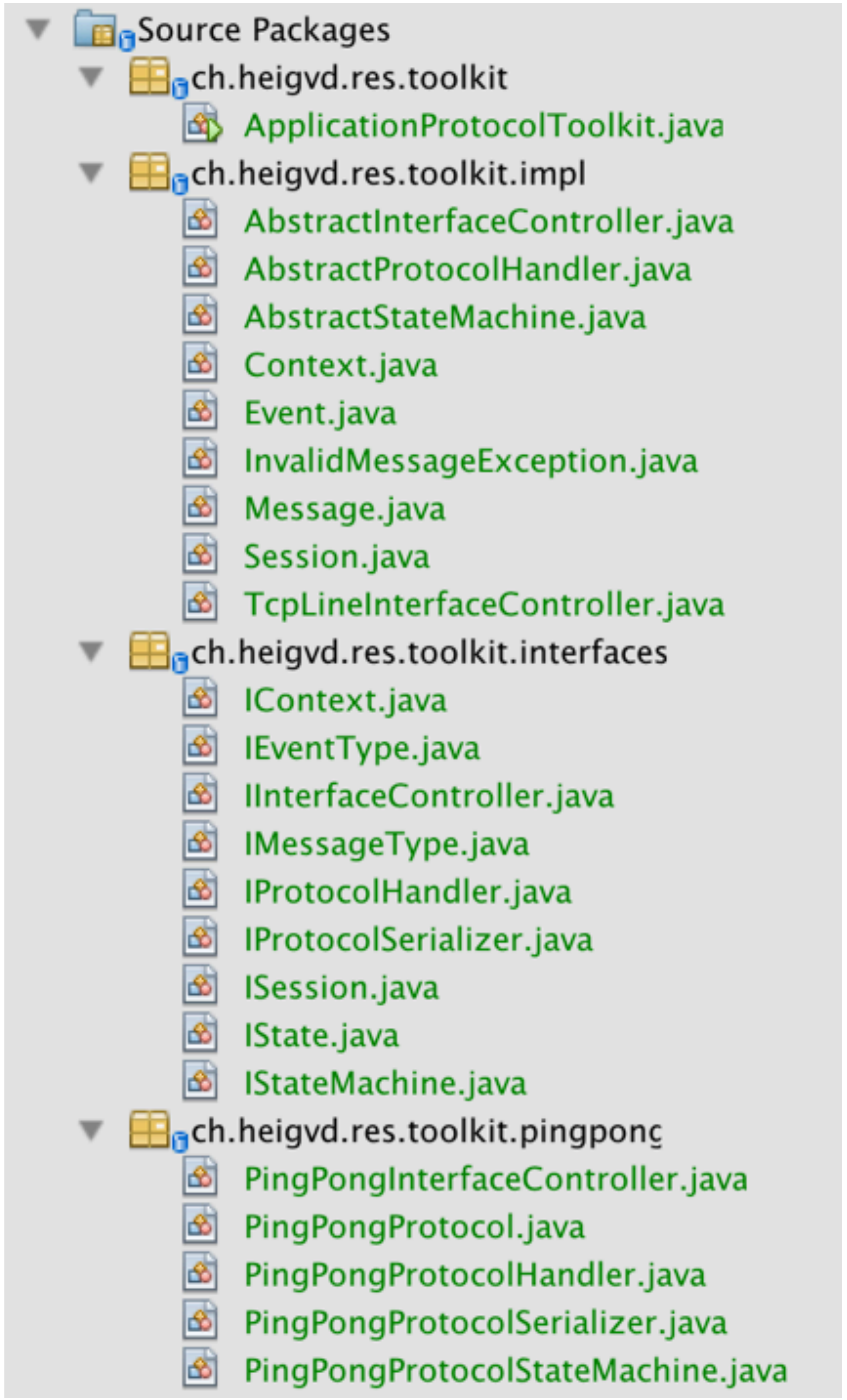
```java
public interface IProtocolHandler {
     /**
      * This callback is invoked when the interface controller has detected that
      * a new session has been started.
      * @param sessionId the id of the new session
      * @param context a context object, which will be passed to the state machine
      * so that it can send back messages via the interface controller
      */
     public void onSessionStarted(long sessionId, IContext context);
     /**
      * This callback is invoked when the interface controller has detected that
      * a session has been closed. This might be the case because of explicit
      * events (e.g. termination of a TCP connection) or because of timing considerations
      * @param sessionId the id of the session that has been closed
      */
     public void onSessionClosed(long sessionId);
     /**
      * This callback is invoked when a new message has arrived and needs to be
      * processed (which typically means notifying the protocol state machine)
      * @param sessionId the session to which the message belongs
      * @param message the incoming message
      */
     public void onMessage(long sessionId, Message message);
     /**
      * This callback is invoked when data has arrived but cannot be deserialized
      * into a valid protocol message
      * @param sessionId the id of the session on which the data has arrived
      * @param e the exception thrown during the deserialization process
      */
     public void onInvalidMessage(long sessionId, InvalidMessageException e);
     /**
      * One responsibility of the classes implementing this contract is to provide
      * the protocol-specific class that is responsible for convert wire-level data
      * into application-level messages, and vice versa
      * @return a protocol specific implementation of the IProtocolSerializer interface
      */
     public IProtocolSerializer getProtocolSerializer();
}
```

```java
public interface IProtocolSerializer {

     /**
      * This method converts raw data (obtained at the transport level) into
      * an application-level message. This is used for incoming messages.
      *
      * @param data raw data
      * @return the corresponding application-level message
      * @throws ch.heigvd.res.toolkit.impl.InvalidMessageException
      */
     public Message deserialize(byte[] data) throws InvalidMessageException;

     /**
      * This method converts an application-level message into wire-level data.
      * This is used for outgoing messages.
      *
      * @param message an application-level message
      * @return the corresponding raw data
      */
     public byte[] serialize(Message message);

}
```

```java
public interface IStateMachine {

     /**
      * This method must be called after the state machine has been registered by
      * the ProtocolHandler
      */
     public void init();

     /**
      * This callback is invoked when an event has been notified. There are different
      * types of events: the reception of a protocol message, the signal from a timer
      *
      * @param event
      */
     public void onEvent(Event event);

     /**
      * This callback is invoked after the state machine has transitioned into a new state.
      * @param state the new state for the state machine
      */
     public void onStateEntered(IState state);

     /**
      * This callback is invoked before the state machine transitions into a new
      * state.
      * @param state the old state for the state machine
      */
     public void onStateExit(IState state);

     /**
      * This method is called after the state machine has been unregistered by
      * the ProtocolHandler. Used to destroy resources and stop timers.
      */
     public void destroy();
}
```

The Ping Pong Protocol


TCP connection opened
Event: Command "ping", "help" or "score" has been received
STATE_START
Event: Command "ping", "help" or "score" has been received
Event: Command "ping", "help" or "score" has been received
Event: Command "ping" has been received
STATE_PING
STATE_PONG
Event: Command "pong" has been received
Event: Command "bye" has been received
Event: Command "bye" has been received
STATE_END
TCP connection closed



```
Source Packages
  ch.heigvd.res.toolkit
    ApplicationProtocolToolkit.java
  ch.heigvd.res.toolkit.impl
    AbstractInterfaceController.java
    AbstractProtocolHandler.java
    AbstractStateMachine.java
    Context.java
    Event.java
    InvalidMessageException.java
    Message.java
    Session.java
    TcpLineInterfaceController.java
  ch.heigvd.res.toolkit.interfaces
    IContext.java
    IEventType.java
    IInterfaceController.java
    IMessageType.java
    IProtocolHandler.java
    IProtocolSerializer.java
    ISession.java
    IState.java
    IStateMachine.java
  ch.heigvd.res.toolkit.pingponc
    PingPongInterfaceController.java
    PingPongProtocol.java
    PingPongProtocolHandler.java
    PingPongProtocolSerializer.java
    PingPongProtocolStateMachine.java
```

$\longrightarrow$ Main class

$\longrightarrow$ Abstract and base classes

$\longrightarrow$ Interfaces

$\longrightarrow$ Ping Pong Protocol classes

```java
public static void main(String[] args) {

    // We will use a particular communication interface to interact with peers.
    // (the inteface may rely on TCP, UDP but maybe also on HTTP, E-MAIL, etc.)
    IInterfaceController interfaceController = new
              PingPongInterfaceController(PingPongProtocol.DEFAULT_PORT);

    // We will exchange "raw" serialized data on an interface. Therefore, we need
    // a class to take care of the serialization/deserialization of this raw data
    // from/into application-level messages
    IProtocolSerializer protocolSerializer = new PingPongProtocolSerializer();

    // We use a protocol to communicate with other parties. We need a class to
    // be responsible for the semantics of the protocol (the class knows what
    // needs to be done when certain messages are received via a communication
    // interface
    IProtocolHandler protocolHandler = new PingPongProtocolHandler(protocolSerializer);

    // We need the inteface controller to be connected to the protocol handler,
    // so that messages arriving on the communication interface can be processed
    // by the protocol handler, and so that the results produced by the protocol
    // handler can be sent back via the interface controller
    interfaceController.registerProcotolHandler(protocolHandler);

    // We are ready, so let us start the interface controller and accept incoming
    // messages
    interfaceController.startup();
}
```