The MapleNet Compute Engine Application Programming Interface

The MapleNet Compute Engine Application Programming Interface

Copyright

1 MapleNet Compute Engine Application Programming Interface

MapleNet can be used to harness the mathematical expertise and computational power of the Maple engine to build a range of services that can run on anything from a mobile device to the cloud. This document contains the following:

- A detailed description of the MapleNet compute engine API
- An example communicating with MapleNet from Python

1.1 Communicating with MapleNet

MapleNet uses Google's Protocol Buffers[1] as the interchange format of messages between itself and clients. Protocol buffer enables MapleNet to communicate with clients written in a variety of languages via an independent manner that is simple and efficient in terms of both size and time. Given the protocol buffer definition file listed in Appendix 1.2 a user could compile it to generate accessors in C++, Python, JavaScript and many other programming languages. A detailed description of compiling a protocol buffer definition is also included in the Appendix.

Communication with MapleNet is achieved through the Request and Reply protocol buffer message types as defined in Listings 1 and 2 respectively.

Listing 1: Protocol buffer definition of a MapleNet Request message.

```
message OutputOptions {
    enum Type {
        TEXT = 0;
        MATHML = 1;
    }
    optional Type type = 1      [default=TEXT];
}

message Command {
    oneof command
    {
       string maple = 1;
    }
}
```

```
message PlotOptions {
    enum Type {
        IMAGE = 0;
        PROTOBUFFER = 1;
    }
    optional Type type = 1              [default=IMAGE];
    optional  uint32  pixel_width = 2    [default=400];
    optional  uint32  pixel_height = 3   [default=400];
}

message Session {
    optional string id = 1;
    optional  uint32  timeout = 2;
}

message Request {
    repeated Command commands = 1;
    optional Session session = 2;
    optional PlotOptions plot_options = 3;
    optional OutputOptions output_options = 4;
}
```

The `commands` field of the `Request` message conveys the Maple statements to be evaluated. Multiple statements can be embedded in a single string or as individual strings in the protocol buffers list structure. For example, all three of the following lists of commands will return the same results:

- `["a:=b^2; b:=4; a;"]`
- `["a:=b^2; b:=4;", "a;"]`
- `["a:=b^2;", "b:=4;", "a;"]`

`MapleNet` will parse the `commands` list into individual Maple statements to be evaluated. The outputs generated by each input statement are returned in a manner that is easy to associate.

`Request` messages contain options that specify the format of the output generated by `MapleNet`. The `OutputOptions` message defines an enumeration, `Type`, that dictates if simple text (`TEXT`) or MathML (`MATHML`) output should be returned. Furthermore, the `PlotOptions` message dictates if a GIF image (`IMAGE`) should be returned for plots generated by Maple or if a protocol buffer encoded plot structure (`PROTOBUFFER`). `PlotOptions` also allow the user to specify the size of the plot by the `width` and `height` fields.

Listing 2: Protobuffer definition of a compute service `Reply` message.

```
message Result {
    oneof Type {
        string text = 1;
        string mathml = 2;
    }
}

message MCSEvent {
    oneof event {
        Command command = 1;
        string input = 2;
        Result result = 3;
        string printf = 4;
        string pretty_print = 5;
        string lprint = 6;
        string error = 7;
        string warning = 8;
        string server_error = 9;
        bytes image_plot = 10;
        MapleCloud.Plot plot = 11;
    }
}

message Reply {
    optional Request request = 1;
    repeated MCSEvent event_stream = 2;
}
```

The `MCSEvent` message type is the central element of a `Reply` message. It is modeled as an event stream where each `MCSEvent.command` is parsed by the Maple engine into individual statements and reported as `MCSEvent.input` events. Each `MCSEvent.input` event when evaluated can generate multiple output events that are encapsulated as one of the following event types:

`result` [`Result`] The result returned by the Maple engine in either `text` or `mathml` depending which is selected by setting the value of the `type` field in the `Request` message.

`printf` [`string`] Output from a call to `printf`().

`pretty_print` [`string`] Output from a call to `print`().

lprint [string] Output from a call to lprint().

error [string] Evaluation of the input statement caused the Maple engine to return an error. Refer to the Maple help system to diagnose the specific error (https://www.maplesoft.com/support/help/).

warning [string] As above, but a warning.

server_error [string] An error generated by MapleNet and not the Maple engine. This is typically due to a malformed Request message or an internal server error.

image_plot [bytes] A GIF image is generated for each Maple plot() call.

plot [MapleCloud.Plot] A protocol buffer encoded Maple plot data structure. Currently, this is for for internal use only.

For example, the sequence of Maple code,

```
pow := proc( b, n ) prod := b^n; return prod; end proc; pow(2,4);
```

would cause the following Request message to be generated (as reported from C++ with a call to DebugString()):

```
commands {
  maple: "pow := proc( b, n ) prod := b^n; return prod; end proc;
          pow(2,4);"
}
plot_options {
  type: IMAGE
}
output_options {
  type: TEXT
}
```

MapleNet will parse the string in the commands.maple field into two individual Maple statements; the definition of the procedure pow and the call to it, pow(2,4). MapleNet will then schedule evaluation of the statements on an idle engine from a pool of available Maple compute engines. The Reply message that MapleNet would respond with is the following[1]

[1]In the following examples long commands are broken across multiple lines. The \character is used to indicate the continuation of the command on the next line. This is supported in many terminals so copying and pasting the entire command should work. If that does not work, duplicating the command as a single line by omitting these characters and joining the argument should also work.

```
request{
  commands {
    maple: "pow := proc( b, n ) prod := b^n; return prod; end proc; \
            pow(2,4);"
  }
  plot_options {
    type: IMAGE
  }
  output_options {
    type: TEXT
  }
}
event_stream {
  command {
    maple: "pow := proc( b, n ) prod := b^n; return prod; end proc; \
            pow(2,4);"
  }
}
event_stream {
  input: "pow := proc( b, n ) prod := b^n; return prod; end proc;"
}
event_stream {
  warning: "‘prod‘ is implicitly declared local to procedure ‘pow‘"
}
event_stream {
  result {
    text: "pow := proc (b, n) local prod; prod := b^n; \
           return prod end proc"
  }
}
event_stream {
  input: "pow(2,4);"
}
event_stream {
  result {
    text: "16"
  }
}
```

To eliminate ambiguity each `Reply` message contains a copy of the original `Request` message that `MapleNet` received.

The principle element of the `Reply` message is the `event_stream` list. This records the temporal series of typed events that were generated by the Maple engine while evaluating the commands submitted in the `Request` message. For example, upon commencing evaluation of the previous `Request` message the command string is the initial event injected into the stream. After which the first parsed statement to be evaluated from the command string is injected as an `event_stream.input` event (the procedure definition). Following each input event a series of output events resulting from the evaluation of the input statement are reported [2]. Two events are generated by the procedure definition. First, a `warning` is reported by the Maple engine that the variable `prod` is not defined and will be interpreted as being locally defined in `pow` (the declaration was purposely omitted to illustrate that a single input statement can generate multiple output events). The second output event reported in the `event_stream` is the `result` of instantiating the procedure `pow`. Aside from multithreaded examples all output events from an input statement are reported after the given `event_stream.input` and before the next input statement is listed in the event stream. Accordingly, the second input statement parsed from the command string is now reported in the event stream, the call to `pow(2,4)`. Finally, the result of `"16"` which is the result from of evaluating the procedure `pow` with arguments `(2,4)` is evident in the stream.

1.2 Python Interaction

This section presents a simple Python client script that receives some Maple code as user input, sends a `Request` message to `MapleNet` for it to evaluate, waits for the `Reply` message and then reports the results. The complete source for the example can be found in
`<MapleNetInstallDir>/examples/pythonMNSEx.py`. For convenience, the necessary code generated by the protocol buffer compiler is also included in the examples directory.

[2] Note that the output generated by statements terminated with "`:`" are omitted as defined by the Maple programming language

Listing 3: A python example that communicates with `MapleNet` via the `Request` and `Reply` messages to evaluate Maple code.

```
#!/usr/bin/python
import argparse
import httplib
import urlparse

# Import the protobuffer definitions of the Reply and Request messages.
import MapleComputeService_pb2 as MCS

class ComputeEngine:

    # Create a connection to MapleNet at serverURL (http://host:port)
    def __init__( self, serverURL ):
        urldata = urlparse.urlsplit( serverURL )
        self.connection = httplib.HTTPConnection( urldata.hostname, urldata.
            port )

    # Create a MapleComputeService.Request protobuffer message that can
    # be sent to MapleNet for evaluation.
    def formatMCSMessage( self, commands ):
        mcsMsg = MCS.Request()
        mcsMsg.plot_options.type = MCS.PlotOptions.IMAGE
        mcsMsg.output_options.type = MCS.OutputOptions.TEXT
        mapleCommand = mcsMsg.commands.add()
        mapleCommand.maple = commands
        msgStr = mcsMsg.SerializeToString()
        return msgStr

    # Retrieve the MapleComputeService.Reply message returned from MapleNet.
    def getMCSReply( self ):
        response = self.connection.getresponse().read()
        mcsMsg = MCS.Reply()
        mcsMsg.ParseFromString( response )
        return mcsMsg

    # Request that MapleNet evaluate the passed in commands.
    def mcsEval( self, commands ):
        msg = self.formatMCSMessage( commands )
        self.connection.request( "POST", "/maplenet/mnserver/mcs/", msg )
        return self.getMCSReply()

# Fetch the passed in URL where MapleNet is running.
parser = argparse.ArgumentParser( prog="pythonMNSEx", description='A python
    example interacting with MapleNet.' )
parser.add_argument( '--mns', default="http://localhost:8080" )
args = parser.parse_args( )

# Establish a connection to MapleNet.
compute = ComputeEngine( args.mns )

while True:
    commands = raw_input( "\n\nEnter Maple statements (press Enter to
        evaluate, ^c to quit): " )

    # Request that MapleNet evaluate the commands entered.
```

```
52      reply = compute.mcsEval( commands )
53
54      # Print the reply from MapleNet.
55      for e in reply.event_stream:
56          if e.HasField( "input" ):
57              print( "> {0:s}".format(e.input) )
58          elif e.HasField( "result" ):
59              if e.result.HasField( "text" ):
60                  print( "\t\t\t{0:s}".format( e.result.text ) )
61              elif e.result.HasField( "mathml" ):
62                  print( "\t\t\t{0:s}".format( e.result.mathml ) )
63          elif e.HasField( "debug" ):
64              print( "DGB> {0:s}".format( e.debug ) )
65          elif e.HasField( "printf" ):
66              print( "{0:s}".format( e.printf ) )
67          elif e.HasField( "pretty_print" ):
68              print( "\t\t\t{0:s}".format( e.pretty_print ) )
69          elif e.HasField( "lprint" ):
70              print( "{0:s}".format( e.lprint ) )
71          elif e.HasField( "error" ):
72              print( "{0:s}".format( e.error ) )
73          elif e.HasField( "warning" ):
74              print( "{0:s}".format( e.warning ) )
75          elif e.HasField( "server_error" ):
76              print( "{0:s}".format( e.server_error ) )
```

The functions of interest in the example are `formatMCSMessage()`, `getMCSReply()`, and `mcsEval()`. The function `formatMCSMessage()` on lines 16-25 illustrates the creation of a `Request` message to be sent to `MapleNet`. On line 19 a new `Request` message is allocated followed by initialization of the output option fields. The `Maple` code to evaluate is then added to the `commands.maple` field and finally the message is serialized in preparation for transmission to `MapleNet`.

Sending the `Request` message to an instance of `MapleNet` (running on port 8080 of the localhost) is accomplished in the function `mcsEval`. To connect to a server running on a different host the `-mns` option can be passed to the script. First a `Request` message is created and then on line 37 it is posted to the endpoint `/maplenet/mnserver/mcs`. Finally, it waits for a `Reply` message to be return from `MapleNet`.

Another sequence of code to take of note of is lines 55 to 76. Here we see an example of indexing into the `event_stream` inside of a `Reply` message. This follows the idiom of checking for the existence of a field via the `HasField()` method prior to attempting to access the date field.

References

[1] Google. Protocol Buffers Version 2 Language Specification. https://developers.google.com/protocol-buffers/docs/reference/proto2-sp%ec/.

Appendix

MapleNet Protocol Buffer Definition

This section provides the full protocol buffer definition of the MapleComputeService.Request and MapleComputeService.Reply message types as well as instructions on how to compile and incorporate them into an application.

The first step in using the message types is to download and install Google's protocol buffer that is available at https://developers.google.com/protocol-buffers/. After doing so, you will be able to use the protocol buffer compiler, protoc, to compile the definition files located in <MapleNetInstallDir>/include/. For example, the following command can be used to compile all of the files in the include directory and place the output in the user created directories cppBuildDir and pythonBuildDir.

```
sh-4.1$ mkdir cppBuildDir pythonBuildDir
sh-4.1$ protoc --proto_path=<MapleNetInstallDir>/include/ \
               --cpp_out=cppBuildDir \
               --python_out=pythonBuildDir \
               <MapleNetInstallDir>/include/*.proto
```

This will generate both C++ and Python source files that can be incorporated into user applications that interact with MapleNet. Note that you will need to add the generated source located in cppBuildDir and pythonBuildDir to your build infrastructure.

Referring to the Python example previously discussed in Listing 3, on line 7 the message types defined in the generated source file MapleComputeService_pb2.py are imported for use within the client.

Complete definition of the protocol buffer `Request` and `Reply` messages.

```
syntax = "proto2";
package MapleComputeService.Compute;
option go_package = "maplenetserver/maplecomputeservice";
import "Plot.proto";

message OutputOptions {
    enum Type {
        TEXT = 0;
        MATHML = 1;
    }
    optional Type type = 1      [default=TEXT];
}

message Command {
    oneof command {
       string maple = 1;
    }
}

message PlotOptions {
    enum Type {
        IMAGE = 0;
        PROTOBUFFER = 1;
    }
    optional Type type = 1              [default=IMAGE];
    optional uint32 pixel_width = 2     [default=400];
    optional uint32 pixel_height = 3    [default=400];
}

message Session {
    optional string id = 1;
    optional uint32 timeout = 2;
}

message Request {
    repeated Command commands = 1;
    optional Session session = 2;
    optional PlotOptions plot_options = 3;
    optional OutputOptions output_options = 4;
}

message Result {
    oneof Type {
```

```
        string text = 1;
        string mathml = 2;
    }
}

message MCSEvent {
    oneof event {
        Command command = 1;
        string input = 2;
        Result result = 3;
        string printf = 4;
        string pretty_print = 5;
        string lprint = 6;
        string error = 7;
        string warning = 8;
        string server_error = 9;
        bytes  image_plot = 10;
        MapleCloud.Plot plot = 11;
    }
}

message Reply {
    optional Request request = 1;
    repeated MCSEvent event_stream = 2;
}
```