Oktopous CCXML PIK 1.4

User manual

Phonologies (India) Private Limited

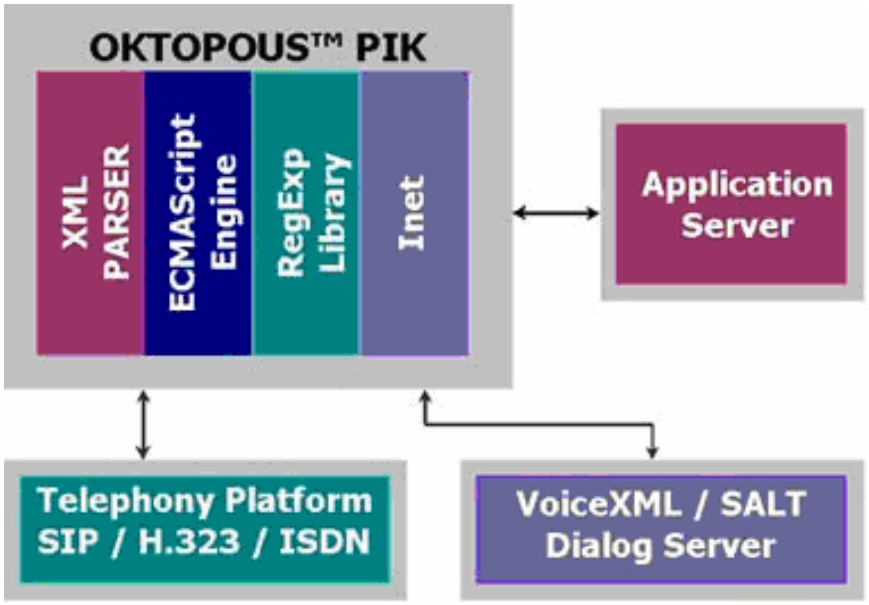
Table of Contents

1. Introduction to OktopousPIK

Oktopous PIK is a C++ based ccxml Interpreter that can be integrated with any telephony platform to provide call controlling functionality as per W3C's last call working draft for ccxml (April 2010).

This document describes various components of Oktopous PIK and also the interfaces provided by it to interface with various telephony platforms. This document also gives you an insight in how to integrate Oktopous PIK with a telephony platform using the interfaces. Provided are code snippets where required.

2. Architecture



3. Oktopous PIK Interfaces:

3.1 OktoInterpreter

OktoInterpreter is the interface that facilitates interaction between Platform and the ccxml Interpreter. If the platform wants to initiate, destroy or use a ccxml Interpreter instance, it must use the OktoInterpreter interface.

The following is the list of functions provided by this interface.

1. Initialize

```
static bool Initialize(OktoLog * log, OutputQueue   **output, EventThreadPolicy policy);
```

This function is used by platform to initialize the Interpreter. It initializes some of the resources needed for Interpreter like OutputQueue and the logging interface. This also sets the threading policy, but this function does not instantiate Interpreter. This function is called only once.

It takes as argument a pointer to an **OktoLog** resource, address of a pointer to an instance of **OutputQueue**, and the **ThreadPolicy** which the platform wishes to use. The OutputQueue is used by Interpreter to send any command it wishes to be acted upon by the platform. Value of ThreadPolicy can be one of **POLICY_DEFAULT** and **POLICY_POOLED** .Description of the policies can be found in **Components of the Interpreter.**

This function is called only once when the platform initializes.

2. DeInitialize

```
static void Deinitialize(OutputQueue *output );
```

This function is used by the platform to shutdown the Interpreter. This function destroys the OutputQ associated with Interpreter instance concerned. This function takes as argument address of the **OutputQueue** associated with the interpreter you wish to deinitialize. This function is also called only once when the platform is shutting down.

3. CreateInstance

```
static OktoInterpreter *CreateInstance();
```

This function lets Platform create a real instance of Interpreter in the memory. It takes no argument and returns a pointer to the created instance of Interpreter.

4. DestroyInstance

```
static void DestroyInstance( OktoInterpreter *interp );
```

This function lets Platform destroy an instance of Interpreter. It takes as argument the address of the **OktoInterpreter** instance the platform wishes to destroy.

5. Run

```
virtual Result Run( const VXIchar  * initialURI,
                    const VXIchar  * sessionid,
                    const VXIchar  * fetchid ,
                    const VXIMap   * args ,
                    Resources      * resources )
```

This function assigns the resources to Interpreter. It also fetches the ccxml file specified by initialURI parses it, and according to the threading policy runs a thread/threads for event processing.

This function takes as argument a pointer to the **URI** of a ccxml document, a unique sessionId for this ccxml Session, a pointer to **fetchid (if URI not given)** of ccxml document fetched earlier, a pointer to a VXIMap **args** and a pointer to an instance of structure **Resources**, which has the resources needed to run an Interpreter Session. The fourth argument VXIMap *args is supposed to be Option properties that affect interpreter behavior. It is right now used by the platform to send parent session id to the child Interpreter session being invoked, but it can be used for sending any such data to an Interpreter session being invoked.

6. PostEvent

```
virtual Result PostEvent( VXIMap *event, VXIunsigned delay )
```

This function lets the platform post telephony events to input queue of Interpreter. It takes as argument address of the VXIMap which contains the event and delay in milliseconds, delay specifies the interval after which the platform wishes the event to be processed.

7. CancelEvent

```
virtual bool CancelEvent(int sendid)
```

This function lets the platform cancel sending of an event which was sent using <send>. The platform takes the CancelEvent out of OutputQueue checks the sendId and removes the event matching this sendId from the queue of events to be executed.

3.2 OktoInet

OktoInet interface enables the platform to interact with the Fetch component of Interpreter and thus facilitates downloading of CCXML applications using http/https protocol. It supports both asunchronous and synchronous fetching. It defines following methods

1. Fetch

```
virtual InetResult Fetch( const VXIchar *  fetchid,
                          const VXIchar *  uri,
                          const VXIMap  *  properties,
                          VXIMap        ** event )
```

Before calling this function you need to fill the VXIMap **properties**, these properties control the kind of download you want. This function takes as argument a pointer to **fetchid**, the **uri** of the file it wants to fetch and pointer to a VXIMap containing **properties** (filled up before),and a pointer to address of a VXIMap **event**. When the function returns, the map **event** has

either **fetch.done** or **error.fetch** and also contains the **fetchid** passed while calling. Here is a list of arguments to fill VXIMap **properties** with:

- Method
- EncryptionType
- TimeOut
- Coockie
- MaxAge
- MaxStale
- **FETCHCALLBACK**

Method can be either GET or POST (like HTTP). MaxAge and MaxStale are used to control the cache behavior of Inet resource. MaxAge specifies the interval after which a cached entry should be considered to be expired. MaxStale suggests the duration upto which an expired entry can be used.

FETCHCALLBACK is relative to async fetch and very important, as it is the way by which the platform can inform Interpreter about completion of a fetch event. First the platform needs to define a function like following:

Following is the signature of function myFetchCallBack platform is supposed to implement a function like this this function should find the InterpreterSession belonging to the sessionId passed,and post this VXIMap fetchEven to the Interpreter.

```
void myFetchCallBack(VXIchar* sessionId, VXIMap* fetchEvent);
```

```
  /*Now in the module which takes commands out from OutputQueues, platform should do
something like following*/

 VXIMap *myEvent = myEventQueue->GetCommand();
 VXIString* evtName = (VXIString *) VXIMapGetProperty(myEvent,L"name";
 VXIString* evtSessionId = (VXIString *) VXIMapGetProperty(myEvent,L"sessionid");

if(evtName==L"command.fetch")
{
        VXIMap *properties;
        properties=VXIMapCreate();

        //Get the pointer to interpreter session using this evtSessionId
        OktoInterpreter *myIntp = platform->getInterpreterFromSessionId(evtSessionId)

        /*In the VXIMap properties set the property FETCHCALLBACK to address of
myFetchCallBack        (Platform's version of fetchcallback)*/

        VXIMapSetProperty(properties,L"FETCHCALLBACK", (VXIValue *)
        VXIPtrCreate((void*)Platform::myFetchCallBack));

        /*In the VXIMap "properties" set the property "FETCHSESSIONID" to sessionid(value
of      sessionId of the Interpreter for which platform is doing this fetch.)*/

        VXIMapSetProperty(properties,L"FETCHSESSIONID",(VXIValue *)
VXIStringCreate(sessionid));


        /*In this form Fetch will be called passing the VXIMap properties*/
        myInet->Fetch(fetchid,myURL,properties,NULL);

}
```

When the Inet subsystem completes the async fetch, it notifies the platform by calling the myFetchCallBack function. Platform should do something similar to this in the call back function.

```
void  Platform::fetchCallBack(VXIchar* mysessionId, VXIMap* fetchEvent) {

     if (mysessionId != NULL) {
          OktoInterpreter *myInterpreter = platform-
>getInterpreterFromSessionId(mysessionId);
          if (myInterpreter != 0) {
               const VXIString *name = (VXIString *)
VXIMapGetProperty(fetchEvent,NAME_FIELD);
               if (name != NULL) {
                    Log("Posting %ls",VXIStringCStr(name));
                    myInterpreter->PostEvent(fetchEvent,0);
                    // Interpreter instance handles the async fetch from here. It parses
the
                    // downloaded content and stores the parsed content in a local
repository
                    // called FetchList. This content can later be used by <goto> or
<createccxml>

                    Log("Posted %ls",VXIStringCStr(name));
               }
          }
     }
}
```

TimeOut specifies the interval in seconds after which a timeout should be thrown if the document can not be downloaded..

2. Initialize

```
static InetResult Initialize(int maxAsync);
```

This function is used by platform to initialize the Inet resource. It takes as argument an integer specifying the number of asynchronous downloaders to initialize.

3. CreateInstance

```
static OktoInet *CreateInstance();
```

This function lets Platform create a real instance of Inet in the memory. It takes no argument and returns a pointer to the created instance of Inet.

4. DestroyInstance

```
static void DestroyInstance( OktoInet *inet );
```

This function lets Platform destroy an instance of OktoInet. It takes as argument the address of the OktoInet instance the platform wishes to destroy.

5. Deinitialze

```
static InetResult Deinitialize();
```

This function is used by the platform to shutdown the Inet Resource and do the necessary cleanup.

3.3 OktoScript

The CCXML standard requires Interpreter to support execution of ECMAScripts .The OktoScript Interface facilitates this execution .It lets the platform create scopes and execution contexts for the script and execute them. Following is the list of function provided by this interface.

1. Initialize

```
static OktoScript::Result Initialize(VXIunsigned diagBase,
                                     OktoLog * log,
                                     VXIlong runtimeSize,
                                     VXIlong contextSize,
                                     VXIlong maxBranches);
```

This function is used by platform to initialize the OktoScript resource. It takes as argument a pointer to OktoLog resource, maximum runtime size the platform wishes to allow, size of context platform wishes to be, and maximum number of allowed nested scopes. The last 3 arguments enable the platform in controlling the amount of main memory OktoScript resource can use.

2. CreateInstance

```
static OktoScript *CreateInstance();
```

This function lets Platform create a real instance of OktoScript resource in the memory. It takes no argument and returns a pointer to the created instance of OktoScript.

3. DestroyInstance

```
static void DestroyInstance( OktoScript * );
```

This function lets Platform destroy an instance of OktoScript resource. It takes as argument the address of the OktoScript instance the platform wishes to destroy.

4. Deinitialze

```
static OktoScript::Result Deinitialize();
```

This function is used by the platform to shutdown the OktoScript Resource and do the necessary cleanup.

5. CreateContext

```
virtual Result CreateContext(OktoScriptContext ** cxt)
```

This function lets the platform create and initialize a new script context. This function takes as argument a pointer to pointer to an object of **OktoScriptContext.** When the function returns with success, it places the address of newly created context in this variable. Currently one context is created per thread.

6. DestroyContext

```
Result DestroyContext(OktoScriptContext ** cxt)
```

This function allows the platform to destroy an execution context with the necessary cleanup. This takes as argument address of a pointer to an existing instance of **OktoScriptContext**, destroys it and does the required cleanup.

7. CreateVarExpr

```
virtual Result CreateVarExpr(OktoScriptContext * cxt,
                             const VXIchar *name,
                             const VXIchar *expr)
```

This function lets the platform create a variable relative to scope and in a given context, and then initializes the variable. This function takes as argument address of the **OktoScriptContext** instance, you want to create a variable within the name of the variable to be created ,and the expression to be evaluated and then assigned as the initial value to the variable. If this third argument is empty or NULL, the initial value of the variable is set to **ECMAScript undefined**.

8. CreateVrValue

```
virtual Result CreateVarValue(OktoScriptContext * cxt,
                              const VXIchar *name, const VXIValue *value)
```

This function lets the platform create a variable relative to scope and in a given context, and then initializes the variable. This function takes as argument address of the **OktoScriptContext** instance you want to create a variable within the name of the variable to be created ,and the **Value** to be assigned as the initial value. If this third argument is NULL, the initial value of the variable is set to **ECMAScript undefined**.

9. SetVarExpr

```
Result SetVarExpr(OktoScriptContext *cxt,
                  const VXIchar * name, const VXIchar *expr)
```

This function lets the platform set a script variable to an expression relative to current scope. This function takes as argument, address of the OktoScriptContext instance you want to create a variable within the name of the variable to be set and an ECMAScript expression to be evaluated and assigned to the variable.

10. SetReadOnly

```
virtual Result SetReadOnly(OktoScriptContext *cxt,
                           const VXIchar *name)
```

This function lets the platform set a variable read-only relative to a scope. This function takes as argument address of the **OktoScriptContext** instance, and the name of the variable within the context which has to be set as read-only.

11. SetVarValue

```
virtual Result SetVarValue(OktoScriptContext * cxt,
                           const VXIchar *name,
                           const VXIValue *value)
```

This function lets the platform set a script variable to an expression relative to current scope. This function takes as argument, address of the **OktoScriptContext** instance you want to create a variable within the name of the variable to be set and a **VXIValue** to be assigned to the variable.

12. GetVar

```
virtual Result GetVar(const OktoScriptContext *cxt,
                      const VXIchar *name, VXIValue **value)
```

This function lets the platform get the value of a variable in a script context. It takes as argument address of the OktoScriptContext to get the variable from, name of the variable and third argument is address of a VXIValue instance which is set to the value of the variable when the function returns.

13. CheckVar

```
virtual Result CheckVar(const OktoScriptContext *cxt, const VXIchar *name)
```

This function lets the platform check if a variable in a OktoScriptContext is defined or not. This function takes as argument address of an OktoScript instance and the name of the variable you wish to check for. If a variable has the value NULL it is considered to be defined or specifically **ECMAScript Defined**.

14. Eval

```
virtual Result Eval(OktoScriptContext *cxt,
                    const VXIchar * name, VXIValue ** res)
```

This function enables the platform in executing a script in a OktoScriptContext, it also returns the result of execution if required. This function takes as argument address of the OktoScriptContext instance to execute the script within the name of the buffer which contains the script text and address of a VXIValue pointer. After execution the result is placed in this third argument, if the result is not desired NULL should be passed as third argument.

15. PushScope

```
virtual Result PushScope(OktoScriptContext *cxt,  const VXIchar *name,
                         const OktoScript::ScopeType type)
```

This function lets the platform add a nested scope to an OktoScriptContext. This function takes as argument address of the OktoScriptContext to add the new scope to, name of the scope and type of Scope to be added. The name of the scope is used to allow referencing variables from an explicit scope within the scope chain, such as "myscope.myvar" to access "myvar" within a scope named "myscope".
The third argument can take one of 2 values NATIVE_SCOPE or ALIAS_SCOPE. NATIVE_SCOPE instructs the function to create a real scope and link it to existing scope chain, while ALIAS_SCOPE instructs the function create an alias for the currently active scope.

16. PopScope

```
virtual Result PopScope(OktoScriptContext *cxt)
```

This function lets the platform pop a context from the scope chain (remove a nested scope). This function takes as argument the address of an OktoScriptContext and removes the topmost nested scope from it.

17. ClearScopes

```
virtual Result ClearScopes(OktoScriptContext *cxt)
```

This function lets the platform Reset the scope chain to the global scope. It takes as argument address of the OktoScriptContext to pop scopes from and pops all the nested scopes.

18. GetName

```
virtual std::string GetName() ;
```

This function returns the name of the interface/subsystem it is being called in.(in this case OktoScript).

3.4 OktoLog

The OktoLog interface enables Oktopous to print diagnostic and error logs.

1. CreateInstance

```
 static OktoLog* CreateInstance();
```

This function lets Platform create a real instance of OktoLog resource in the memory. It takes no argument and returns a pointer to the created instance of OktoLog.

2. DestroyInstance

```
static void DestroyInstance(OktoLog *);
```

This function lets Platform destroy an instance of OktoLog resource. It takes as argument the address of the OktoLog instance the platform wishes to destroy.

3. Diagnostic

```
virtual void Diagnostic( const VXIchar *  sessionID,
                         const VXIchar * subtag,
                         const VXIchar * format, ... ) = 0;
```

The Diagnostic function can be used by platform to print diagnostic logs. Basically it can be used to log and see when events are arriving, when they are being handled and what CCXML element is currently executing etc. This function takes as argument the sessionId of the CCXML session you are printing a log in, subtag, and format string to be used optionally if you wish to print some other messages or values of some internal variables at the time of printing log. So, this function takes variable no. of arguments and after the format string you can specify the names of the variables for which yoy want to print values.

4. Error

```
virtual void Error(const VXIchar *sessionID,
                   const VXIchar * moduleName,
                   VXIunsigned     errorID,
                   const VXIchar * format, ... ) = 0;
```

4. Components of Interpreter

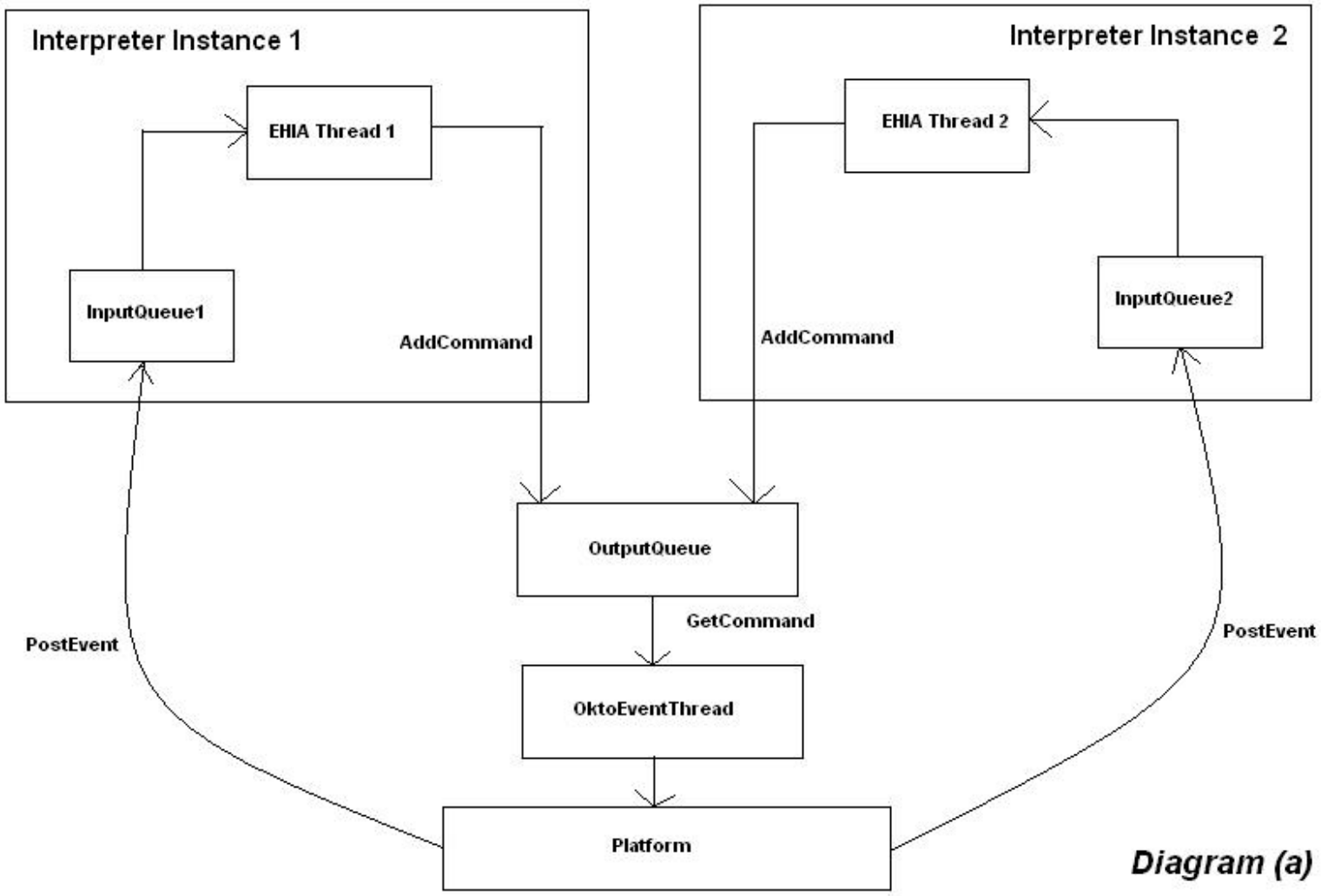


Diagram (a)

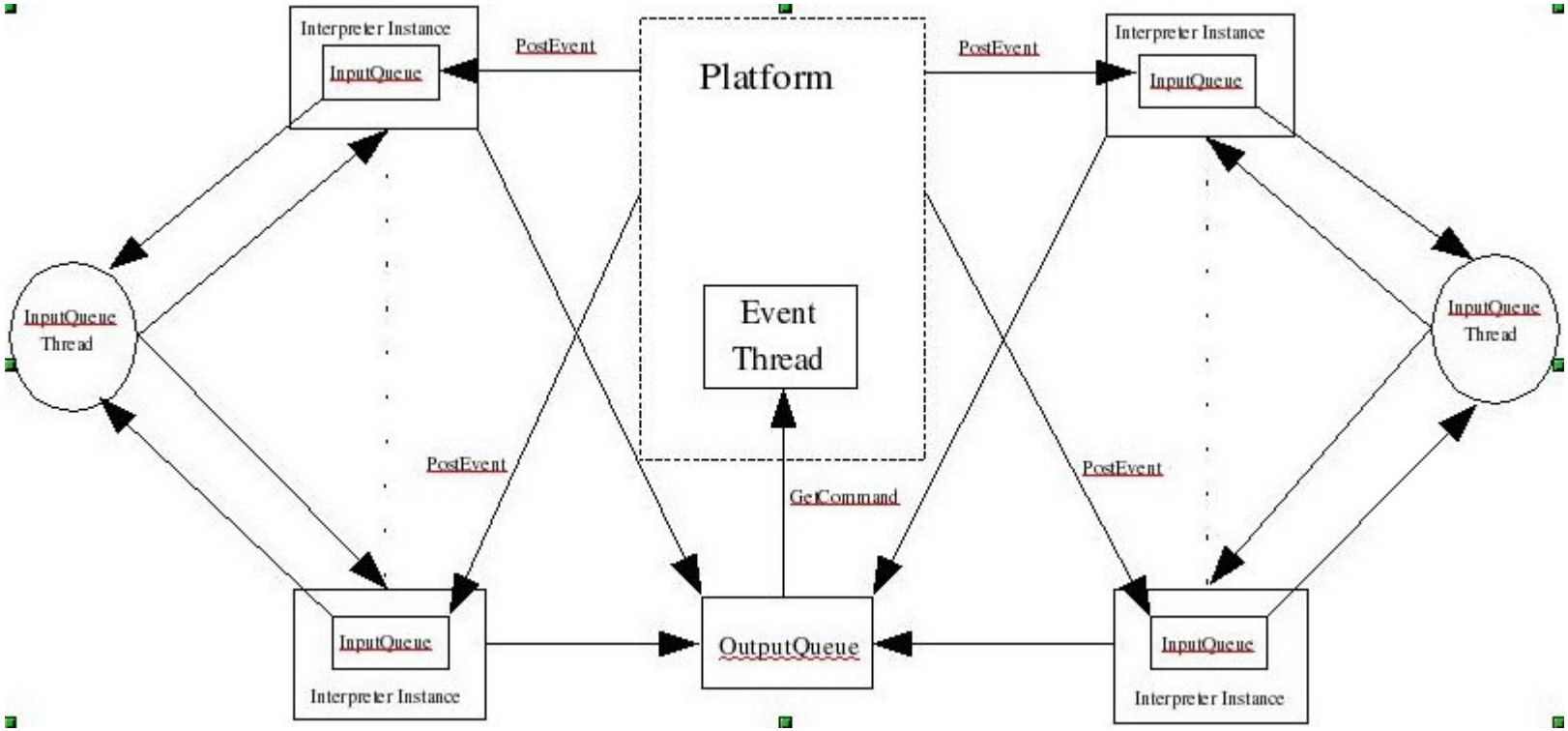


Diagram (b)

4.1 InputQueue

InputQueue is the place where the platform places the telephony events it wishes to send toOktopous. InputQueueThread / EHIAThread reads this queue and gives it to Oktopous. The InputQueue is of type **EventQueue**,which has been implemented as a **Priority Queue**, which uses **Time** as its priority. Each Queue entry consists of an **Event*** and **TimeStamp T**. This queue is sorted in ascending order of the time (T). T is the time when the event was generated + any delay specified for that event.

If the **ThreadPolicy** is **POLICY_DEFAULT**, each InputQueue has 1 thread (**EHIAThread**) monitoring it while if the **ThreadPolicy** is **POLICY_POOLED**, a number of **InputQueue**s are monitored by a single thread (**InpuQueueThread**). Please find the details in the sub-section **ThreadPolicy**.

4.2 Output Queue

OutputQueue is the Interface for handling commands issued by the interpreter.Whenever an element is encountered, it is converted into a command and sent to the OutputQueue. Platform is expected to monitor this queue, and execute the commands from this queue.

So OutputQueue is the place where the **Oktopous** places **commands** which it wishes to be acted upon by **platform**.e.g. suppose Oktopous encounters an **<accept>** element inside a transition element,and it has to execute it,it simply makes an output command **(Implemented as VXIMap) "command.accept"** and places it in the **outputQueue**.Now the **platform** will take it from **OutputQueue** and depending upon the telephony protocol(e.g. SIP) it will take action needed for accepting a call.

4.3 Thread Policy

ThreadePolicy lets the **platform** choose 1 of the 2 available implementations of **EHIA**.This actually affects the way **InputQueue** takes events from **platform** and the way **InputQueue** gives it to the **Oktopous**.There are 2 ThreadPolicies available, **POLICY_DEFAULT** and **POLICY_POOLED**. If you choose **POLICY_DEFAULT** ,the architecture of the implementation will be like **diagram (a)**.In case of **POLICY_POOLED** it will be like **Diagram (b)**.

POLICY_DEFAULT

In case of POLICY_DEFAULT the architecture of Interpreter is like **Diagram(a)**.In this case each Interpreter has an **InputQueue** and an **EHIAThread** monitoring the InputQueue associated with itself. There is a common **OutputQueue** for all instances of **Interpreter**. Using a call to **Interpreter→PostEvent()** the **Platform** posts events in the **InputQueue** of the desired **Interpreter Instance**. EHIAThread of that Instance takes it out from **InputQueue** and processes the event,according to the ccxml file. Output is generated,(Output is nothing but the actions Interpreter wishes to be taken by **platform**) **EHIAThread** writes this output to **OutputQueue** using **OutputQueue–>AddCommand()**.

Platform has a thread **OktoEventThread** which continuously monitors the OutputQueue and as soon as some command comes ,it takes it out using a call to **OutputQueue→GetCommand()**.This command also has the **SessionId** of the ccxml session it belongs to,so that Platform can know which Interpreter session this command belongs to.

In this case for every new **Interpreter session** (or say for every new **call**) 1 new thread is generated which monitors the **InputQueue**.If you have thousands of calls running this can be a overhead for the **platform**.

So there is a choice if you do not want to generate 1 new thread for every new call, you can use **POLICY_POOLED** as your **ThreadPolicy**.

POLICY_POOLED

In case of **POLICY_POOLED** the architecture of the implementation is like **Diagram(b)**. In this case we do not have individual threads for each **Interpreter** (or specifically **InputQueue**). With the initialization of OktoInterpreter we initialize a ThreadPool, When a call comes or a **CCXML session** starts we call **threadPool–>AddInputQueue (Interpreter *)**. This associates the given Interpreter's InpuQueue with one of the **InpuQueueThread**s of **ThreadPool**. Each **InputQueueThread** maintains a list of pointers to **Interprete**rs, it is monitoring InputQueues of. It iterates through the list monitoring each queue processing the events ready to be processed and writing the Output to **OutputQueue** from where Platform takes it using **OktoEvenThread** and takes the desired action. Because it uses lesser threads, it does not cause the overhead which can be caused by the POLICY_DEFAULT implementation. However in this case InputQueues does not have luxury of own EHIAThread at service, so this may take more time (some more milliseconds)to process events in comparison to the POLICY_DEFAULT implementation.

4.4 Fetcher

Fetcher is responsible for downloading the ccxml document. It supports **http** and **https** URIs. It is also used for **synchronous** and **asynchronous fetches** of documents. In case of asynchronous fetch it parses the document after fetching and maintains the repository of parsed document which can be accessed and used for starting new CCXML sessions using the **fetchid** of the document.

4.5 Transition validator

During CCXML session when an event arrives we need to check what state we are in and then we need to find the transition in the CCXML application matching to this combination of state and event. This task is done by TransitionValidator.

The event name inside a **<transition>** can be a regular expression too. Then we need to compile this regular expression and then we match it with a string. Compilation of regular expressions is a CPU intensive task. **TransitionValidator** implements an algorithm to minimize this compilation. So every time we ask TransitionValidator to find a matching transition, following steps are taken:

1. Check if state,evnt and condition attributes for the transition are empty, in that case it is an automatic transition.

2. Then check if the current transition element has already been processed. If not, then process it now i.e. compile the regular expression inside evnt attribute and store it in a data container for later use. This time we do not have to compile regular expression every time that event is fired. If this transition element is already processed, we skip this step.

3. Match the compiled regular expression with the current event.

5. Components of Inet

5.1 Tornado

Tournado implements a HTTP/1.1 client supporting HTTP/HTTPS/FILE protocols. It used the CURL HTTP client library for protocol level operations. It Interacts with Cache Manager to optimize fetching.

5.2 Cache Manager

Cache manager stores the downloaded files in a local repository. It uses LastMofied, If-Not-Modified and Cache-Control headers of the HTTP responses to decide whether a file should be returned from cache or to be downloaded from internet. Cache manager also supports maxage and maxstale parameters as required by the CCXML Specification.

5.3 Async Fetcher

When Inet susbsystem is initialized it creates a Thread pool of Downloaders. The number of Downloaders can be configured when initializing the Inet resource. These Downloaders accept Async fetch requests from the Interpreter and notify the Interpreter sessions when the fetch is complete. Async Fetcher is not directly accessed by the Interpreter. But it interacts with the implementation platform to notify the completion of a fetch.

6. Components Of Scripter

Scripter is basically the implementation of the OktoScript interface. This enables execution of javascript inside ccxml applications as required by W3C's last call working draft for CCXML. The Scripter uses the SpiderMonkey JavaScript engine. There is one runtime instance. Each instance of **Interpreter** creates a new **context**, and inside this **context** it pushes scopes as and when needed. There is a limit on total amount of main memory that scripter can use. This is specified while initializing Scripter using **OktoScript::Initialize()**.

7. The Integration WalkThrough

7.1 Platform implements the OktoPlatform Interface provided by Oktopous.

7.2 Platform **instantiates** OktoLog resource and then **initializes** different resources needed for Interpreter,

```
//Instantiate OktoLog resource
OktoLog* myLog=OktoLog::CreateInstance();

//Initialize OktoInet Resource with 2 async downloaders
Oktonet::InetResult res = OktoInet::Initialize(2);

//Initialize OktoScript resource with myLog as the Logging resource,
//runtime size of 1024*1024 bytes,contextSize 128*1024 bytes ,and
//maximum number of allowable branches set to 10.
OktoScript::Initialize(myLog,1024*1024,128*1024,10);

//Initialize Oktopous with myLog logging resource,outQ (the
 //OutputQueue to be sent commands to),and with thread policy
//POLICY_DEFAULT
OktoInterpreter::Initialize(myLog,&outQ,OktoInterpreter::POLICY_DEFAULT);

OR

//Initialize Oktopous with myLog logging resource,outQ (the OutputQueue to
//be sent commands to),and with thread policy POLICY_POOLED
OktoInterpreter::Initialize(myLog,&outQ, OktoInterpreter::POLICY_POOLED);
```

7.3 Platform should implement a thread which will continuously monitor the **OutputQueue** and inform platform about incoming commands. Platform should run this thread.

7.4 Now a new instance of **Interpreter** should be created , when a call comes, or platform wishes to place an outbound call, but before creating this instance platform needs to **instantiate** all the resources needed.

```
OktoInterpreter::resources = new struct OktoInterpreter::Resources;

//Instantiate OktoInet resource
myInet = OktoInet::CreateInstance();

//Instantiate OktoScript resource
myECMA = OktoScript::CreateInstance();

//Instantiate OktoInterpreter
myInterpreter = OktoInterpreter::CreateInstance();

/*fill the resource structure of Platform*/
 resources->inet = myInet;
 resources->jsi = myECMA;
 resources->log = myLog;
 resources->platform = this;
```

7.5 Before running the eventProcessing thread you need to generate a unique **sessionid.** This gives you a way to be able to differentiate between different calls.

```
//Generate a unique ID for this session and assign it to mySessionId
VXIchar *mySessionId=genSessionId();
```

7.6 Before running the eventProcessing thread you need to generate a unique **sessionid.** This gives you a Now run the Event Processing thread, Pass it the ccxmlUrl and the unique id you generated along with the resource structure initialized in step 4.

```
/*Run the EventProcessing Thread,Pass it the ccxmlUrl "myurl",sessionid "mId" of this
session,third argument is "NULL" becaues we are passing URL(either ccxml document URL or  fetchId
of a prefetched ccxml Document has to be passed),
Fourth argument is "NULL" becuase this is not a child Interpreter session(Had it been a child
session we would set a VXIMap,with the property "parentid" set as essionid of the parent" and a
pass a pointer to this VXIMap here"),the last
argument passed is a pointer to the structure "resource" which we filled earlier in step 4.*/

myInterpreter->Run(myUrl,mId,NULL,NULL,resources);
```

7.7 Whenever the platform needs to post an event it has to create a VXIMap, and set the name of the event and sessionID, and the attributes required by the event(if required) and then call PostEvent. For example if platform wants to post **"connection.alerting"**, it would do it as follows:

```
VXIMap* evt=VXIMapCreate();

VXIMapSetProperty(evt,L"connection",VXIValueClone((VXIValue*)connectionObject));

VXIMap *event=VXIMapCreate();

/*Set the "name" property of VXIMap "event" to VXIchar * "connection.alerting"*/

VXIMapSetProperty( event, L"name" , reinterpret_cast<VXIValue*>
(VXIStringCreate(L"connection.alerting")) );

/*Set the "sessionid" property of VXIMap "event" to VXIchar * "sessionId"*/
VXIMapSetProperty( event, L"sessionId" , reinterpret_cast<VXIValue*>
(VXIStringCreate(sessionId) ));

/* Set the 'connectionid' property of VXIMap 'event' to VXIchar* 'connid' */
VXIMapSetProperty(event,L"connectionid", reinterpret_cast<VXIValue *>
(VXIStringCreate(connid)));

//Create a VXIMap for connection object
VXIMap* connectionObject=VXIMapCreate();

//Set properties that need to be available inside session.connection
//For example setting of 'local' and 'remote' properties is shown here
```

```
VXIChar* localStr=L"Some_value_for_local";
VXIChar* remoteStr=L"Some_value_for_remote";
VXIMapSetProperty(connectionObject, L"local",(VXIValue*) VXIStringCreate(localStrl));
VXIMapSetProperty(connectionObject,L"remote",(VXIValue*)VXIStringCreate(remoteStr));

//The connection object map should be inserted inside the event map
VXIMapSetProperty(event,L"connection",VXIValueClone((VXIValue*)connectionObject));
//Post Event connection.alerting with zero delay
myInterpreter->PostEvent(event,0);

/* Note:
 * Assuming the connectionid value is 'c1' ,the connection object inside ccxml
 * script can be accessed as session.connections['c1'].
 *
 * The above discussion applies to dialog and conference objects as well.
 * The property local will be available as session.connections['c1'].local
 * For a list of properties supposed to be passed in connection/dialog/conference
 * maps please refer CCXML spec.
 */
```

7.8 The thread implemented by platform which monitors the OutputQueue can take commands out from OutputQueue, as these commands are of type VXIMap, any properties set earlier using VXIMapGetProperty can be retrieved using VXIMapSetProperty. Now the thread can give it to Platform for further action desired. Lets take an example of command.accept (which is generated in response to <accept> element in ccxml file.

```
VXIMap *event = myEventQueue->GetCommand();
VXIString* evtName = (VXIString *) VXIMapGetProperty(event,L"name";
VXIString* evtSessionId = (VXIString *)
VXIMapGetProperty(event,L"sessionid");
if(evtName==L"command.accept")
{
        /*Accept the call,this is implemented by platform and may
        depend upon telephony protocol in use and many other things.*/

platform->accept();
```

/*Call has been accepted now, so post connection.connected to the Interpreter Session.*/

```
VXIMap *newEvent=VXIMapCraete();
/*set the "name" property of VXIMap "newEvent" to VXIchar * "connection.connected".*/
VXIMapSetProperty( newEvent, L"name" , reinterpret_cast<VXIValue *>
                  (VXIStringCreate(L"connection.connected")) );

/*set the "sessionid" property of VXIMap "newEvent" to VXIchar * "sessionId"*/

VXIMapSetProperty( newEvent, L"sessionId" , reinterpret_cast<VXIValue *>
                   (VXIStringCreate(sessionId) ));

/*You need to know now which interpreter session this event should be posted to.This
has to be implemented by platform, should take sessionId as input and return pointer
to the Interpreter instance related to this sessionId.*/

OktoInterpreter* mInterpreter = platform->getInterpreterPointer(evtSessionId);
//Post Event connection.connected with zero delay.
mInterpreter->PostEvent(newEvent,0);
```

7.9 When ccxml application exits,Interpreter instance calls the platform→CCXMLComplete().and platform can free the resources used and other clean-up operations.

```
/*CCXML session has exited,so its time to clean up everything*/
//This function is a virtual function in the class OktoPlatform
// and must be defined by the platform implementor.

platform->CCXMLComplete;
```