Cisco Remote Expert Interactive Controller SDK Developer Guide

Release 1.9.4

March 5, 2015

Note All advertising materials mentioning features or use of this software must display the following acknowledgement: "*This product includes software developed by the University of California, Berkeley and its contributors.*"

Overview

This document provides information about the Cisco Remote Expert Interactive Controller (REIC) software development kit (SDK). The REIC SDK allows Cisco Remote Expert (RE) to be integrated into customer applications.

Topics in this guide include:

- Introduction, page 2
- Acronyms, page 2
- High Level Component Interaction, page 3
- SDK Component Details, page 4
  - REIC APIs, page 4
  - Call Control, page 5
  - Utility Control, page 6
  - Job Control, page 6
  - RESC Communicator API, page 12
  - REM Event Manager APIs, page 13
  - Cobra Manager APIs, page 16
  - Common API, page 17
- Co-Browsing, page 19
- SIP Call Broadcaster, page 20

- IEC Device API, page 20
- SDK Configuration, page 21
- Notification and Polling, page 22
- REIC Properties, page 22
- Logging, page 23
- SDK Folder Structure, page 23
- Minify REIC SDK, page 24
- Minimum Requirements, page 24
- Sequence Diagrams for Components, page 24
  - Call Manager API, page 25
  - Feedback Manager API, page 26
  - Event Manager API, page 27
  - Jobs, page 28
- Error Details, page 35
  - API Related Errors, page 35
  - RESC Related Errors, page 36
- Fail Over, page 37
- Adding the REIC SDK to a Client Application, page 39

Introduction

REIC SDK contains the core control logic of REIC, which segregates controller code from the client user interface (UI) code. Previously control logics code were integrated with the client UI code written in Flex. With the help of this SDK client, UI code can be independent from the core control logic. As a result, the SDK can be integrated with client applications easily without the need of Flash.

This document describes the high-level design diagrams and application programming interface (API) details for each operation performed in REIC.

Acronyms

The following acronyms are used in this guide:

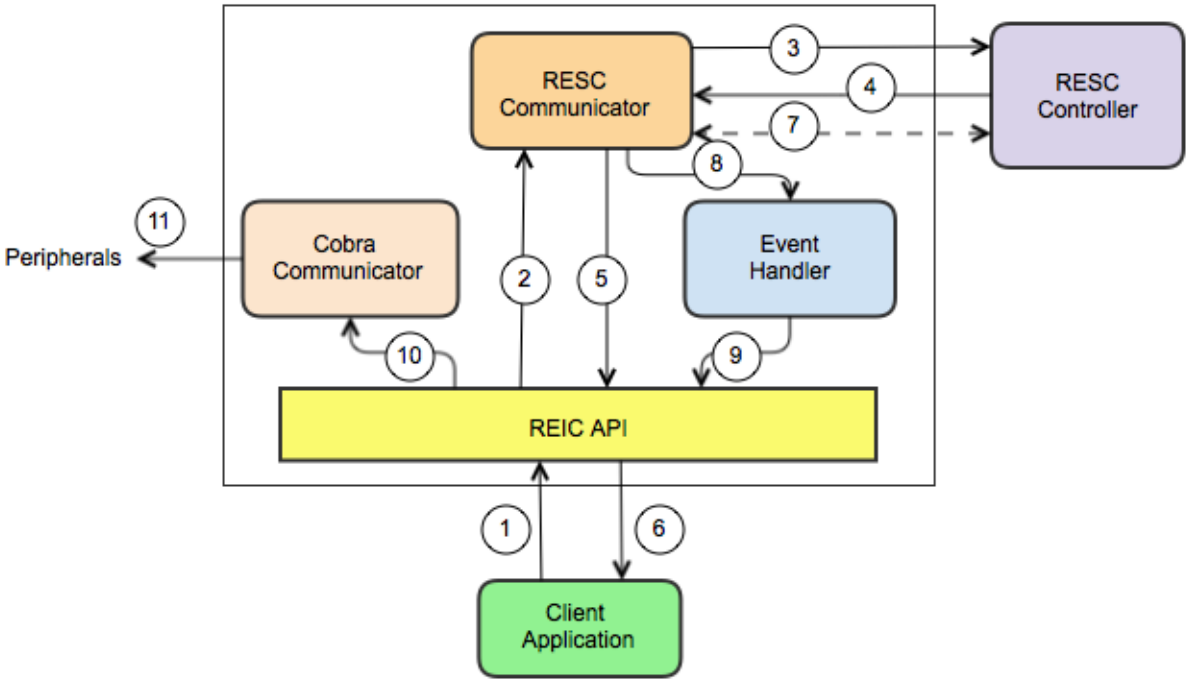
- API – Application Programming Interface
- DN – Directory Number
- eREAD – eRemote Expert Agent Desktop (uses Cisco Finesse)
- IEC – Interactive Experience Client
- IEM – Interactive Experience Manager
- RE – Remote Expert
- REAC – Remote Expert Administration Console
- READ – Remote Expert Agent Desktop (uses Cisco Agent Desktop)

- REIC – Remote Expert Interactive Applications Control
- REM – Remote Expert Manager
- RESC – Remote Expert Session Controller
- SDK – Software Development Kit
- TP – Cisco TelePresence
- UI – User Interface
- VNC – Virtual Network Computing

High Level Component Interaction

The figure below illustrates the internal communication between components. Refer to the list below the figure for explanations of each interaction.

Figure 1 SDK Modules Interaction



1. Client application invokes the REIC API (e.g., StartCall(), EndCall(), Print(), Scan(), etc.).
2. REIC API communicates with RESC with the help of the RESC Communicator. In the case of starting a call, for example, the IEC serial number along with expert Directory Number (DN) are passed to RESC to initiate a call.
3. RESC Communicator forwards the request to RESC.
4. RESC sends response back to the RESC Communicator (i.e. session details when a call is established).
5. RESC Communicator forwards the response to REIC API.

6. Proper call back function is invoked in the client application depending on response.
7. Web Socket communication or polling call established between RESC Communicator and RESC (e.g., active call session) to get active session details.
8. During the active call, any event (e.g., call on hold, print, scan) from RESC is detected by Event Handler.
9. Event Handler triggers REIC API depending on the event type.
10. All communications related to print, scan, document camera, etc. are handled between REIC API and Cobra Communicator.
11. Cobra Communicator sends corresponding signals or commands from the client application to the respective peripherals.

SDK Component Details

The main class for accessing all the APIs is 'REIC'. This is the global namespace, which encapsulates all the properties and methods related to different parts of the SDK. All common tasks are defined in the API in SDK, which is common and used across other modules of the SDK. The common API can be accessed by calling the '*REIC.common*' namespace.

After loading the REIC SDK, the client application needs to initialize the SDK to load the REIC property file. This step is mandatory. For details information on initialization, refer to the *SDK Configuration* section of this guide.

Details of the following APIs of REIC SDK are presented below:

- REIC APIs including call control, utility management and job control APIs
- RESC Communicator API
- REM Event Manager API
- Cobra Manager APIs
- Common API

REIC APIs

REIC APIs are classified as call control, utility management, and job control APIs. Call control APIs are used to manage REM calls whereas utility management APIs are used to manage session feedback. Job control APIs are used to manage various jobs such as print, signature capture, document camera, scan, co-browsing, etc.

Proper call back functions need to be passed from client applications to get the response details after invoking each API. The response status can be either 'SUCCESS' or 'FAILURE'. On success, the argument of a successful callback function contains the response details. The response detail is embedded in a JSON object returned by the API and can be accessed by '`response.json`' where `response` is the argument passed to the callback function. In case of error, response details can be accessed by the argument passed to the error callback function.

Status from REM server after invoking web service can be accessed by '`response.json.statusCode`'. It can be either 'SUCCESS' or 'FAILED'.

Callback Response Structure:

- Success

  - Status: `response.status`
  - JSON (contains response details): `response.json`
- Failure
  - Status: `response.status`
  - Error Code (if any): `response.errorCode`
  - Error Message (if any): `response.errorMessage`

Call Control

The 'REIC' namespace contains the 'callManager' property, which is an object containing the basic call operation methods including startCall([arguments]), endCall([arguments]), getSessionStatus([arguments]), etc.

1. Start REM Call – Invoking the following API starts REM call:

   ```
   REIC.callManager.startCall(iecSerial, expertTypeId, customerId,
   successCallBackFn, failCallBackFn)
   ```

   - `iecSerial` – IEC's serial number
   - `expertTypeId` – Expert type identification
   - `customerId` – Customer identification [number]
   - `successCallBackFn` – Callback function provided by the client application for success
   - `failCallBackFn` - Callback function provided by the client application for failure
2. Disconnect or End REM Call – Invoking the following API disconnects REM call:

   ```
   REIC.callManager.endCall(iecSerial, successCallBackFn,
   failCallBackFn)
   ```

   - `iecSerial` – IEC's serial number
   - `successCallBackFn` – Callback function provided by the client application for success
   - `failCallBackFn` - Callback function provided by the client application for failure
3. Session Status or Call Status – Session status can be received by invoking the following API:

   ```
   REIC.callManager.getSessionStatus (sessionId, successCallBackFn,
   failCallBackFn)
   ```

   - `sessionId` – Session identification
   - `successCallBackFn` – Callback function provided by the client application for success
   - `failCallBackFn` - Callback function provided by the client application for failure
4. Get Session Id – Session identification can be queried using the following API:

   ```
   REIC.callManager.getSessionId()
   ```

   - return – Session identification
5. Set Session Id – Session identification can be set using the following API:

   ```
   REIC.callManager.setSessionId(sessionId)
   ```

   - `sessionId` – Session identification
   - return – Session identification

Utility Control

Session Feedback – 'REIC' namespace contains 'sessionFeedback' property to access session feedback related methods.

1. Get Feedback List – Session feedback questions master list can be fetched from the server by calling the following API:

   ```
   REIC.sessionFeedback.getQuestionAnswerList (locale, iecSerial,
   successCallBackFn, failCallBackFn)
   ```

   - `locale` – Locale code
   - `iecSerial` - IEC's serial number
   - `successCallBackFn` – Callback function provided by the client application for success
   - `failCallBackFn` - Callback function provided by the client application for failure

2. Submit Feedback – Session result feedback can be submitted by invoking the following API:

   ```
   REIC.sessionFeedback.submitFeedback(sessionId, feedback,
   successCallBackFn, failCallBackFn)
   ```

   - `sessionId` - REM session identification
   - `feedback` – Array of objects. Each object contains questionId and answerId pair for multiple choice questions or questionId and answerText pair for text type questions.
   - `successCallBackFn` – Callback function provided by the client application for success
   - `failCallBackFn` - Callback function provided by the client application for failure

   Sample feedback parameter:

   ```
   feedback = [{questionId: 1, answerId: 1}, {questionId: 1,
   answerText: 'I am text'}]
   ```

Job Control

Job can be controlled by an expert (i.e. expert triggered job) or a customer (i.e. customer controlled job) in REM. The following table highlights the owner of each job supported in REM.

Table 1 Job Control Information

Job	Expert/Agent	Customer
Print	Yes	No
Scan	Yes	Yes
Document Camera	Yes	No
Card Reader	No	Yes
Signature Capture	No	Yes
VNC	Yes	No

All the APIs are described here for each job irrespective the job owner. The jobs that can be controlled by an agent or customer are described in the *REM Event Manager APIs* section.

Print

Printer APIs can be accessed by the 'REIC.printer' property.

1. Initialize Printer – Calling the following API initializes the attached printer. It internally calls Cobra browser APIs. The API returns either 'true' or 'false'.

   ```
   REIC.printer.init()
   ```

   - return – `true/false`

2. Start Print – Calling the following API starts printing the document. It internally calls Cobra browser APIs.

   ```
   REIC.printer.start(printUrl, successCallBackFn, failCallBackFn)
   ```

   - `printUrl` – URL of the document to be printed
   - `successCallBackFn` – Callback function provided by the client application for success
   - `failCallBackFn` - Callback function provided by the client application for failure

   If it is customer triggered printing, job status must be updated by client application by invoking the following method defined in common APIs in SDK:

   ```
   REIC.common.updateJobStatus(sessionId, jobStatus, jobType,
   documentId, message, successCallBackFn, failCallBackFn)
   ```

   - `jobStatus` and `jobTyp`e can be passed with the help of following APIs:
     a. `REIC.common.jobStatus.<status>` for jobStatus [e.g., for print, `status` can be 'INITIATED', all statuses are defined in common API]
     b. `REIC.common.jobType.<jobname>` for jobType [e.g., for print, job name is 'PRINT', all job types are defined in common API]
   - `message`: Success message or failure message (i.e. reason for the error)

   Note For details about common API, refer to *Common API* section of the document.

Scan

1. Get Scanner – Calling the following API can query the attached scanning device:

   ```
   REIC.scanner.getScanner()
   ```

   - return – List of scanners as an array. If no scanner is found, 'null' is returned.

2. Set Scanner – Scanner can be set by calling the following API. The name must be valid and passed as an argument to the method.

   ```
   REIC.scanner.setScanner(name)
   ```

   - `name` – Name of the scanner (must be a string)
   - return – `true/false`

3. Get DpiX – The API for reading DpiX is the following:

   ```
   REIC.scanner.getDpiX()
   ```

   - return – value as integer

4. Set DpiX – Invoking the following API sets the DpiX:

   ```
   REIC.scanner.setDpiX(newDpiX)
   ```

- `newDpiX` – new dpix as integer
- return – `true/false`

5. Get DpiY – The API for reading DpiY is the following:

   `REIC.scanner.getDpiY()`

   - return – value as integer

6. Set DpiY – Invoking the following API sets the DpiY:

   `REIC.scanner.setDpiX(newDpiY)`

   - `newDpiY` – new dpiy as integer
   - return – `true/false`

7. Get Color Mode – The API for reading color mode is the following:

   `REIC.scanner.getColor()`

   - return – `true/false`

8. Set Color Mode – Invoking the following API sets the color mode:

   `REIC.scanner.setColor(enable)`

   - `enable` – 'true' or 'false'
   - return – `true/false`

9. Get Document Source – The API for reading document source is the following:

   `REIC.scanner.getSource()`

   - return – document sources ('null' if no sources found)

10. Set Document Source – Invoking the following API sets the document source:

    `REIC.scanner.setSource(name)`

    - `name` – new source (must be string)
    - return – `true/false`

11. Initialize Scanner – Calling the following API initializes the scanner:

    `REIC.scanner.init(onErr, onFin)`

    - `onErr` – callback function for scan error. This callback function is invoked by Cobra API. Error message is passed as an argument of this callback.
    - `onFin` – callback function for scan success invoked by cobra API
    - return – `true/false`

12. Start Scanner – The following API starts scan:

    `REIC.scanner.start()`

13. Stop Scanner – The following API stops scan:

    `REIC.scanner.stop()`

14. Get Scanned Document – The scanned document can be accessed with the help of the following API:

    `REIC.scanner.getScannedDocument()`

    - return – scanned document in base64 data format

15. Send Scanned Document – The scanned document can be sent to RESC with the help of following API:

```
REIC.scanner.sendScanDocument(sessionId, imageType,
base64ImageData, successCallBackFn, failCallBackFn)
```

- `sessionId` – Session identification
- `imageType` – Type of image
- `base64ImageData` – Base64 image data
- `successCallBackFn` – Callback function provided by the client application for success
- `failCallBackFn` - Callback function provided by the client application for failure

16. Send Command Acknowledgement – Commands sent to the IEC are acknowledged when invoking the following API:

```
REIC.scanner.sendAcknowledgeCommandExecute(sessionId, jobId,
commandId, successCallBackFn, failCallBackFn)
```

- `sessionId` – Session identification
- `jobId` – Job identification
- `commandId` – Command identification
- `successCallBackFn` – Callback function provided by the client application for success
- `failCallBackFn` - Callback function provided by the client application for failure

17. Remove events – After using the scanner API, the events can be disconnected from the Cobra scanner API by the following API:

```
REIC.scanner.removeEvents()
```

Note The scanner must be set before initializing the scanner and performing any other operation on the scanner. This step is mandatory; otherwise the scanner will not be operational.

Document Camera

The document camera is initialized after initializing the video encoder with the help of the following API defined in the 'REIC.vidEncoder' namespace:

```
REIC.vidEncoder.init(expertIP)
```

`expertIP – Expert IP`

If the IP address is empty, the default camera host IP address that is configured in the IEM's policy for the document camera is used.

1. Initialize Camera –

```
REIC.docCamera.init(command, commandId, jobId)
```

- `command` – Command sent to the camera
- `commandId` – Command identification
- `jobId` – Job id
- return – `true/false`

2. Check Camera Status – This command can be sent if and only if the document camera is ready. This check is mandatory before sending any other command.

```
REIC.docCam.isDocumentCameraReady()
```

- return – `true/false`

3. .Trigger Command –

   `REIC.docCam.triggerCommand(command, commandId, jobId, fPQRS)`

   - `command` – Command sent by agent
   - `commandId` – Command identification
   - `jobId` – Job identification
   - `fPQRS` – Camera zoom position(p, q, r, s value)

   The above API accepts the command sent by agent and finds the mapping to the actual command data (RS232) defined in the command file. The command file is packaged inside SDK in the following location: 'reicsdk/data/Document Camera/rs232Cmd.json'

   The following commands are supported in REM:

```
"START_CAM"   :  "8101040002FF",
"STOP_CAM"    :  "8101042F03FF,8101040003FF,1",
"START_LASER" :  "8101042F02FF",
"STOP_LASER"  :  "8101042F03FF",
"ZOOM_IN_X"   :  "8101040702FF,8101040700FF,1",
"ZOOM_IN_2X"  :  "8101040702FF,8101040700FF,2",
"ZOOM_IN_4X"  :  "8101040702FF,8101040700FF,4",
"ZOOM_IN_10"  :  "8101040702FF,8101040700FF,10",
"ZOOM_IN_15X" :  "8101040702FF,8101040700FF,15",
"ZOOM_OUT_X"  :  "8101040703FF,8101040700FF,1",
"ZOOM_OUT_2X" :  "8101040703FF,8101040700FF,2",
"ZOOM_OUT_4X" :  "8101040703FF,8101040700FF,4",
"ZOOM_OUT_10X" :  "8101040703FF,8101040700FF,10",
"ZOOM_OUT_15X" :  "8101040703FF,8101040700FF,15",
"PRESET_CAPTURE_1" : "81090447FF",
"PRESET_CAPTURE_2" : "81090447FF",
"PRESET_CAPTURE_3" : "81090447FF",
"PRESET_CAPTURE_4" : "81090447FF"
```

   For details on camera commands refer to the "Setting Document Camera Predefined Zoom Buttons for Agents" section of the *Cisco Remote Expert Manager Administration Guide*.

   Note The commands are sent to the Vaddio document camera with the help of the 'REIC.vaddio' property.

4. Turn Camera On –

   `REIC.docCam.on()`

5. Turn Camera Off –

   `REIC.docCam.off()`

6. Turn Laser On –

   `REIC.docCam.laserOn()`

7. Turn Laser Off –

   `REIC.docCam.laserOff()`

8. Zoom In –

   `REIC.docCam.zoomIn()`

9. Zoom In Slow –

   `REIC.docCam.zoomInSlow()`

10. Zoom Out –

    `REIC.docCam.laserOff()`

11. Zoom Stop –

    `REIC.docCam.laserOff()`

12. Zoom Preset 1 –

    `REIC.docCam.preset1()`

13. Zoom Preset 2 –

    `REIC.docCam.preset2()`

14. Get Zoom Position –

    `REIC.docCam.getZoomPosition()`

Card Reader

1. Initialize Card Reader –

   `REIC.magstripe.init(onScanned, onError, onScanning, onOpened)`

   - `onScanned` – Callback invoked by Cobra API when the card scan is completed
   - `onError` – Callback invoked when there is an error
   - `onScanning` – Callback invoked when card is scanning
   - `onOpened` – Callback invoked when the card reader is opened

2. Close Card Reader –

   `REIC.magstripe.closeCardReader()`

   - return – `true/false`

3. Access Card Data as Object –

   `REIC.magstripe.getParseData()`

   - return – Data as object

4. Print Card Data as String –

   `REIC.magstripe.getParseDump()`

   - return – Data as string

5. Get Card Reader Name –

   `REIC.magstripe.getCardReaderName()`

   - return – Card reader name

Signature Capture

1. Initialize –

   `REIC.signature.init(signaturePad, onMouseDown, onMouseMove, onMouseUp)`

   - `signaturePad` – Canvas DOM element where the signature is drawn
   - `onMouseDown` - Callback function invoked on mouse click on signature pad
   - `onMouseMove` - Callback function invoked on mouse move on signature pad

- `onMouseUp` - Callback function invoked on mouse up on signature pad

2. Adjust Smoothing –

   ```
   REIC.signature.setSmoothing(newSmoothing)
   ```

   - `newSmoothing` – New smoothing value

3. Set Linewidth –

   ```
   REIC.signature.setLineWidth(newLineWidth)
   ```

   - `newLineWidth` – New line width

4. Wipe Signature –

   ```
   REIC.signature.wipeAll()
   ```

5. Send Image –

   ```
   REIC.signature.sendSignature(signaturePad, sessionId,
   successCallBackFn, failCallBackFn)
   ```

   - `signaturePad` – Canvas DOM element where the signature is drawn
   - `sessionId` – Session identification
   - `successCallBackFn` – Callback function provided by the client application for success
   - `failCallBackFn` - Callback function provided by the client application for failure

VNC

Please refer to the *Co-Browsing* section of this guide for API details.

RESC Communicator API

All web service requests are handled by the RESC Communicator component of REIC SDK. On invoking a web service from the REIC API, the request is forwarded to the RESC with the help of the RESC Communicator. The client application can also invoke the methods defined in the RESC Communicator by directly invoking the API.

All web service related methods are defined in the 'rescCommunicator' property of the 'REIC' global namespace.

The API for sending the request for web service can be invoked by calling the following API:

```
REIC.rescCommunicator.getWebService(WebServiceName).invoke(methodN
ame, params, httpMethod, data, successCallBackFn, failCallBackFn)
```

- `WebServiceName` – Name of the web service
- `methodName` – Web service method to be invoked
- `params` – Parameters passed to the web service AJAX call
- `httpMethod` – HTTP method (e.g. 'GET', 'POST', 'PUT', 'DELETE')
- `data` – Data to be sent in post request. For 'GET' request, it must be set to 'null'.
- `successCallBackFn` – Callback function provided by the client application for success
- `failCallBackFn` - Callback function provided by the client application for failure

REM Event Manager APIs

REM Event Manager APIs handle events including call on hold, call connect, call disconnect, and video streaming as well as agent triggered jobs such as print, scan, signature capture, document camera capture, and co-browsing.

All event related APIs are defined under the 'REIC.event' namespace.

Call status polling is started by kiosk notification polling if valid session identification is found; refer to the *Notification and Polling* section of this document for details. Depending on the different call and job status, a particular event is triggered.

The various methods available in event manager API are the following:

1. Trigger Event –

   `REIC.event.trigger(eventName, data)`

   - `eventName` – Name of the event
   - `data` – Call response

2. Attach Event –

   `REIC.event.attach(eventName, handlerFn)`

   - `eventName` – Name of the event
   - `handlerFn` – Function to be invoked on the event

   The client application needs to attach an event to listen for that event. Proper callback function also needs to be passed as the event handler. The argument of the handler function contains the event details.

   The argument of the handler function contains the following object:

   ```
   {
   type: [event type],
   message: [data],
   time: [event time]
   }
   ```

3. Detach Event –

   `REIC.event.detach(eventName)`

   The client application must de-register the event once it is not needed.

In the table below are the various events triggered for the call and jobs (those that are expert triggered) inside the REIC SDK by the Event Manager APIs.

Table 2 Available Events

Event Name	Description
REIC.event.RINGING	Call is ringing
REIC.event.CONNECTED	Call is connected
REIC.event.DISCONNECTED	Call has disconnected
REIC.event.ONHOLD	Call is on hold
REIC.event.ABNORMAL_DISCONNECT	Call has abnormally disconnected
REIC.event.DISCONNECT_BY_TIMEOUT	Call was disconnected by timeout

Event Name	Description
REIC.event.VIDEO_STREAM	Video stream detected. Video list is passed as an event message in the JSON format.
REIC.event.PRINT_INITIATED	Print has started
REIC.event.PRINT_COMPLETED	Print has completed
REIC.event.PRINT_FAILED	Print has failed
REIC.event.SCAN_INITIATED	Scan has initiated
REIC.event.SCAN_COMPLETED	Scan has completed
REIC.event.SCAN_CANCELED	Scan has been canceled
REIC.event.SCAN_DOC_READY	Scanned document is ready to be sent
REIC.event.SCAN_FAILED	Scan has failed
REIC.event.DOCCAM_INITIATED	Document camera capture is initiated
REIC.event.DOCCAM_CANCELED	Document camera capture has been canceled
REIC.event.DOCCAM_COMPLETED	Document camera capture has been completed
REIC.event.SIGNATURE_INITIATED	Signature capture has started
REIC.event.SIGNATURE_CANCELED	Signature capture has ben canceled
REIC.event.VNC_INITIATED	Co-browsing has started
REIC.event.VNC_CANCELED	Co-browsing has been canceled
REIC.event.TP_NOT_ALIVE	TP is not alive

All the above events are triggered inside the REIC SDK.The client application needs to register to the above events to get notification. On an event trigger, particular callback functions are invoked, which have been passed to the attach method during the event registration.

Various jobs handled by the Event Manager APIs are explained below:

1. Print – When an expert initiates print from the agent desktop, the call flow inside the event manager class for print operation is the following:
    1. REIC.event.PRINT_INITIATED is triggered with job details (as JSON) passed to the client in the callback function argument.
    2. Printer is initialized (REIC.print.init()).
    3. Print is started (REIC.print.start(…)).
    4. If printing is successful, the REIC.event.PRINT_COMPLETED is triggered with job details (as JSON) passed to the client in the callback function argument. If printing is not successful (i.e there is a ‘failure’), the REIC.event.PRINT_FAILED is triggered with response message details (as JSON) passed to the client in the callback function argument. For error code, refer to the *Error Details* section of this document.

    For each event, client needs to register to the event with callback functions.
5. Scan –

a. Job owner is kiosk:
    1. REIC.event.SCAN_INITIATED is triggered. Job detail is passed as the argument of callback function.
    2. Set the scanner if scanner is found; else scan operation is completed with error message.

3. Initialize the scanner.
4. Scanner properties are set such as color mode and dpis.
5. Scan is started.
6. If scan fails, the REIC.event.SCAN_FAILED is triggered and the error message is passed in the callback function. For error details, refer to the *Error Details* section of this document. If scan is successful, the REIC.event.SCAN_DOC_READY event is triggered and the following data is passed in the callback function:

```
{
sessionId: [Session id],
base64data: [Base64 image data]
}
```

Client application needs to send the base64data to the expert. For the API to send the image, refer to the Scan API in the *Job Control* subsection.

b. Job owner is agent/expert:

1. Agent sends different scan command to REIC.
2. Command is INIT_SCAN. REIC.event. SCAN_INITIATED is triggered. Job details are passed in the call back function.
3. Command is SCAN_DOC. Scan is started after setting and initializing the scanner.
4. If scan fails, REIC.event.SCAN_FAILED is triggered and the error message is passed in the callback function. For error details, refer to the *Error Details* section of this document.
5. On successful scan the base64 image data is available
6. Command is SCAN_SEND. Image is displayed on READ/eREAD.

7. Document Camera – The document camera job is controlled by the agent. The job is handled by the REM event manager API internally. The following is the operation flow inside SDK:

1. Agent starts camera and camera job is initiated. REIC.event.DOCCAM_INITIATED is triggered with job details in the callback function.
2. Command is START_CAM. Initialize the video encoder. If the document camera is not ready, initialize it; otherwise, send the command to the camera. The camera is started by writing the RS232 command (mapped to START_CAM in command data file) to the Vaddio document camera device with the help of the API defined in the 'REIC.vaddio' namespace.
3. Command is 'STOP_CAM'. Camera is stopped after sending the RS232 command to the Vaddio document camera device.
4. Command is 'PRESET_CAPTURE'. The PQRS value is retrieved from the job response.
5. Command is 'PRESET_EXECUTE'. The PQRS value is sent to the Vaddio document camera device to store the zoom positions.
6. Command is 'PRESET_ADMIN_EXECUTE'. The PQRS value is retrieved from the response and sent to the Vaddio document camera device.

Note For error details, refer to the *Error Details* section of this document.

7. Signature Capture – The following is the operation flow inside SDK:

1. Agent initiates the signature capture.

2. REIC.event.SIGNATURE_INITIATED is triggered. The client application needs to register to this event with the callback function. The job detail is passed to the callback function as an argument.
3. The client application needs to initialize the signature capture using the API defined in the 'REIC.signature' property once the above event is triggered. For details refer to the *Job Control* subsection of this document for signature capture APIs.
4. The client needs to invoke the API for sending the signature to the agent.
5. REIC.event.SIGNATURE_CANCELED is triggered if the signature capture job is canceled by the agent.

Note For error details, refer to the *Error Details* section of this document.

6. VNC – The following is the operation flow inside SDK:
   1. The agent initiates co-browsing (a VNC job).
   2. REIC.event.VNC_INITIATED is triggered with job details.
   3. VNC is started and the response is sent to the agent.
   4. If the agent cancels the VNC job, the REIC.event.VNC_CANCELED is triggered.

Note For error details, refer to the *Error Details* section of this document.

Cobra Manager APIs

Communication between peripherals attached to the IEC and the client application is achieved with the help of the Cobra browser. The Cobra browser API needs to be called in order to handle the devices properly from the client application. Cobra manager exposes a global object called 'global'. All the APIs needs to be invoked on this global namespace.

The communication is managed by 'cobraManager' property of 'REIC' global namespace (`REIC.cobraManager`).

It returns an object containing the 'device' property, which is an object. It contains the various mapping between peripherals and the corresponding cobra APIs:

1. Print –

   `REIC.cobraManager.device.printer`
2. Scan –

   `REIC.cobraManager.device.scanner`
3. Card Reader –

   `REIC.cobraManager.device.cardReader`
4. Document Camera –

   `REIC.cobraManager.device.videoEncoder`

   `REIC.cobraManager.device.serialPort`
5. IEC –

   `REIC.cobraManager.device.serialPort`

6. Application Data –

   `REIC.cobraManager.device.applicationData`

7. VNC (Co-browsing) –

   `REIC.cobraManager.device.vnc`

Common API

The Common API is used across all the modules in the SDK. The Common API is defined under the 'REIC.common' namespace.

The Common API contains a method for sending an AJAX response to the client and validating parameters used in different methods in various APIs. It is an object container for different job types, job statuses, commands, error messages and error codes. Details of various functions inside the 'REIC.common' object are described below.

Note Common API contains various utility methods to validate parameters passed to methods across APIs.

1. Validate Parameters – Parameters passed in various APIs can be validated. The empty, null, or undefined parameter can be checked by the following API:

   `REIC.common.isValidParam(param, message, failCallBackFn)`

   - `param` – Parameter to be checked
   - `message` – Error message if the parameter is not valid
   - `failCallBackFn` - Callback function provided by the client application for failure

2. Check Number Parameter – Parameters passed in various APIs can be checked whether they are numbers or not by invoking this API:

   `REIC.common.isNumber(param, message, failCallBackFn)`

   - `param` – Parameter to be checked
   - `message` – Error message if the parameter is not valid
   - `failCallBackFn` - Callback function provided by the client application for failure

3. Check Valid URL – The following API can be used to validate the URL:

   `REIC.common.isValidUrl(url)`

   - `url` – URL to be checked

4. Send Response – The API to send the response to the client application is the following:

   `REIC.common.sendResponse(successCallBackFn, failCallBackFn, json)`

   - `successCallBackFn` – Callback function provided by the client application for success
   - `failCallBackFn` - Callback function provided by the client application for failure
   - `json` – Response in JSON

5. Job Types – Status for different job types is defined in the 'jobType' object inside the 'REIC.common' object. Job type can be accessed by calling the following API:

   `REIC.common.jobType.<job type>`

- `job type` – Name of the job type. The available job types are PRINT, SCAN, SIGNATURE, and DOCUMENT_CAMERA. For the print operation, for example, call `REIC.common.jobType.PRINT`.

6. Job Status – All job statuses are defined in the 'jobStatus' object inside the 'REIC.common' object. Job status can be accessed by calling the following:

   `REIC.common.jobStatus.<job status>`

   - `job status` – Name of the job status. The available job statuses are INITIATED, CANCELED, COMPLETED, ACKNOWLEDGED, DELETED. For a completed job status, for example, call `REIC.common.jobStatus.COMPLETED`.

7. Error Codes – All error codes can be accessed by calling the following:
   `REIC.common.errorCodes.<name_error_msg>`

   The following error code is available for a print job:

   - PRINT_ERROR - 801

8. Messages – All messages are defined inside the 'messagePool' object inside 'REIC.common'.

9. Call Status – Different call statuses can be accessed by the following object:

   `REIC.common.callStatus`

   The following call statuses are supported:

```
{
RINGING: '0',
CONNECTED: '1',
DISCONNECTED: '2',
ONHOLD: '3',
ABNORMAL_DISCONNECT: '4',
DISCONNECT_BY_TIMEOUT: '5'
}
```

10. Commands – All commands are defined under the 'REIC.common.commands' object.

11. Update Job Status – Job status can be updated by calling the following API:

    `REIC.common.updateJobStatus(sessionId, jobStatus, jobType, documentId, message, successCallBackFn, failCallBackFn)`

    - `sessionId` – Session identification
    - `jobStatus` – Job status
    - `jobType` – Job type
    - `documentId` – Identification of the document
    - `message` – Message for the status (i.e. success or failure message)
    - `successCallBackFn` – Callback function provided by the client application for success
    - `failCallBackFn` - Callback function provided by the client application for failure

12. Update Video Status – Video job status can be updated by calling the following API:

    `REIC.common.updateVideoStatus(sessionId, videoId, status, successCallBackFn, failCallBackFn)`

    - `sessionId` – Session identification
    - `videoId` – Video identification
    - `status` – Video status
    - `successCallBackFn` – Callback function provided by the client application for success

- `failCallBackFn` - Callback function provided by the client application for failure

13. Fetch Kiosk Details – Kiosk details can be fetched from RESC by invoking the following API:

    ```
    REIC.common.getKioskDetails(locale, iecSerial, successCallBackFn, failCallBackFn)
    ```

    - `locale` – Locale code
    - `iecSerial` – IEC's serial number
    - `successCallBackFn` – Callback function provided by the client application for success
    - `failCallBackFn` - Callback function provided by the client application for failure

    Note This API must first be called during the loading of the client application on the kiosk in order to load the expert type details.

14. Check TP Aliveness – The API for checking Cisco TelePresence (TP) aliveness is the following:

    ```
    REIC.common.isTPAlive(iecSerial, successCallBackFn, failCallBackFn)
    ```

    - `iecSerial` – IEC's serial number
    - `successCallBackFn` – Callback function provided by the client application for success
    - `failCallBackFn` - Callback function provided by the client application for failure

Co-Browsing

Co-browsing is achieved via VNC sharing. The VNC server must be started in order to start co-browsing. To enable co-browsing, the client application needs to include Vnc.js in the HTML page.

The following are the APIs for co-browsing:

1. Start VNC server – The following API starts the VNC sever:

    ```
    REIC.vnc.start(password, host, port, readOnly)
    ```

    - `password` – IEC's device maintenance code as the password (optional)
    - `host` – IP address of the IEC device (optional)
    - `port` – Port for VNC server (by default 5980)
    - `readOnly` – Read only access ('true' or 'false')

    After starting the VNC server successfully, the VNC job must be updated with the 'Completed' status (2) using the following API:

    ```
    REIC.common.updateJobStatus(sessionId,REIC.common.jobStatus.COMPLETED, REIC.common.jobType.VNC, '', '')
    ```

    - `sessionId` – Session identification

2. Stop VNC server – To stop the VNC server, invoke the following API:

    ```
    REIC.vnc.stop()
    ```

3. Check Server Status – Invoke the following API to check running status of the VNC server:

    ```
    REIC.vnc.isRunning()
    ```

    - return – `true/false`

4. Read Only Check – Whether the VNC server is accessed as read only can be checked by the following API:

   `REIC.vnc.isReadOnly()`

   - return – `true/false`

SIP Call Broadcaster

Callback functions invoked by the Cobra API for different SIP call statuses (e.g. established, disconnected, error) are defined under the namespace 'REIC.sipBroadcaster'. Callbacks are accessed by invoking the following APIs:

1. Call Established – This method internally starts polling for kiosk notification status. If the poll returns a session identification, another poll for session status is started and kiosk notification poll is stopped.

   `REIC.sipBroadcaster.onEstablished()`

2. Call Disconnected – The following API disconnects and stops any active polling when invoked:

   `REIC.sipBroadcaster.onDisconnected()`

3. Call Error – The following API stops any active polling if there is an error:

   `REIC.sipBroadcaster.onError(code, explanation)`

   - `code` – Error code
   - `explanation` – Reason of the error

IEC Device API

All IEC related APIs are available under the 'REIC.iec' namespace.

1. Initialization – An IEC device can be initialized by invoking the following API:

   `REIC.iec.init(baudRate, dataBits, flowControl, parity, stopBits, failCallBackFn)`

   - `baudRate` – Baud rate
   - `dataBits` – Data bits
   - `flowControl` – Flow control
   - `parity` – Parity
   - `stopBits` – Stop bits
   - `failCallBackFn` - Callback function provided by the client application for failure

   Example:

   ```
   baudRate=BaudRate19200
   dataBits=DataBits8
   flowControl=FlowControlOff
   parity=ParityNonef
   stopBits=StopBits1
   ```

2. IEC Serial – The following API returns the serial number of the IEC device:

   `REIC.iec.getSerial()`

3. Send Command – The following API sends commands to the IEC device:

   `REIC.iec.sendCommand(command)`

   - `command` – The command that needs to be sent

4. Get Screen Index – To get the screen index use the following API:

   `REIC.iec.getScreenIndex()`

   - `return` – Index of the screen

5. Post IEC Reboot – The IEC can be rebooted by invoking the following API:

   `REIC.iec.postIecReboot(successCallBackFn, failCallBackFn)`

   - `successCallBackFn` – Callback function provided by the client application for success
   - `failCallBackFn` - Callback function provided by the client application for failure

   Note The IEC must be rebooted after Cisco TelePresence (TP) comes alive.

6. Get Kiosk Notification Information – To get the kiosk notification information, call the following API:

   `REIC.iec.getKioskNotificationInfo(iecSerial, successCallBackFn, failCallBackFn)`

   - `iecSerial` – IEC's serial number
   - `successCallBackFn` – Callback function provided by the client application for success
   - `failCallBackFn` - Callback function provided by the client application for failure

SDK Configuration

REIC SDK must be initialized after loading the SDK. This step is necessary to load the REIC property file from the REM server or any other server. Once the initialization has completed, the IEC's device serial number and the RESC service URL can be accessed. To initialize SDK, use the following API:

`REIC.config.init(config, successCallBackFn, failCallBackFn)`

- `successCallBackFn` – Callback function provided by the client application for success
- `failCallBackFn` - Callback function provided by the client application for failure
- `config` – Optional configuration object (see example below)

```
config = {
hostname: '10.76.8.190',
port: '8443',
path: 'reic'
}
```

For the above configuration, the REIC property path is: https://10.76.8.190:8443/reic/conf/reic.properties.

The following API can access the IEC's device serial number:

`REIC.config.IEC_SERIAL`

The following API can access the RESC service URL:

`REIC.config.RESC_SERVICE_URL`

The whole property file is stored as an object with key-value pairs describing each property. To access the object invoke the following API:

```
REIC.config.getProperty()
```

Notification and Polling

Notification service is invoked to check for any active REM sessions and TP aliveness once the REIC SDK is loaded. A call to the kiosk notification service returns an active session identification if there is a REM call established. Once the valid session identification is found, polling for session or call status is automatically started to fetch the job details. Intervals for various polling can be configured in the reic.properties file. For detailed information on configuring the property file, please refer to the *REIC Properties* section of this document. The TP aliveness check returns 'true' or 'false'. Once TP comes alive (i.e. it transitions from 'false' to 'true' status), an IEC reboot is performed to restart the IEC device.

Note JavaScript for notification APIs must be loaded after initializing the REIC SDK (`REIC.config.init(config)`).

Polling APIs are defined under the 'REIC.polling' namespace.

1. Kiosk Notification Polling –

   ```
   REIC.polling.startKioskNotificationInfoPoll(interval)
   ```

   `interval` – Interval for polling

2. Session Polling –

   ```
   REIC.polling.startSessionPoll(sessionId)
   ```

   `sessionId` – Session identification

3. TP Aliveness Check Polling –

   ```
   REIC.polling.startTPALiveCheckPoll(interval)
   ```

   `interval` – Interval for polling

4. Notification Polling – Kiosk notification and TP aliveness check polling can be started together by invoking the following API:

   ```
   REIC.polling.startNotificationPoll()
   ```

REIC Properties

The REIC property file is loaded by invoking `REIC.config.init(successCallBackFn, failCallBackFn, config)`. If no config property is provided, the default property file is loaded from the 'conf/reic.properties' location relative to the parent folder of REIC SDK.

The REIC property file contains the following properties:

```
resc.url=https://10.196.97.170:8443/resc/services/
command.kiosk=1!
command.tp=2!

baudRate=BaudRate19200
dataBits=DataBits8
flowControl=FlowControlOff
```

```
parity=ParityNonef
stopBits=StopBits1


# Document camera property
camera.serialPort.device=S0
camera.serialPort.baudRate=9600
camera.serialPort.dataBits=8
camera.serialPort.flowControl=0
camera.serialPort.parity=0
camera.serialPort.stopBits=1

# poll property start
# poll frequency
session.poll=7000
kiosk.notification.poll=5000
tp.poll=10000

# poll flags
kiosk.notification.poll.enable=true
tp.poll.enable=true
# poll property end

#VNC Server
#text | image | none | RE
vnc.server.readonly=false
vnc.server.port=5980
```

Logging

Device level logging (i.e. IEC event logs) is enabled in the API. In case of error in device level logging, the browser level log is enabled automatically. Different types of logging are supported: info, warn, error, and debug. For logging access use the following APIs:

- `REIC.logger.info(text, title)`
- `REIC.logger.warn(text, title)`
- `REIC.logger.error(text, title)`
- `REIC.logger.debug(text, title)`
  - `text` – Message
  - `title` – Title for log context (e.g., logging for scan operation uses ‘SCAN’ as title)

SDK Folder Structure

SDK parent directory is ‘reic_sdk’. The directory contains the following folders:

- ‘src’ for all the source files
- ‘lib’ for libraries used in SDK
- ‘dist’ for minified version of SDK
- ‘doc’ for JS documentation
- ‘conf’ for reic.properties file

- ‘test’ for unit test case code

Minify REIC SDK

REIC SDK is minified using Google Closure Compiler. Since REIC must be initialized manually in the client application before notification poll starts, SDK is minified without the Notification.js file. Notification.js is minified separately. Minified REIC SDK version without notification polling is ‘reic.sdk.min.js and notification.min.js’ for Notification.js.

Note The ‘notification.min.js’ must be loaded after invoking `REIC.config.init([args])`.

To minify the SDK invoke the ‘`ant`’ command in the parent directory of REIC SDK. Minified version of the SDK is placed under the ‘dist’ directory. Build script also creates JS Documentation under the ‘doc’ folder.

Minimum Requirements

The minimum requirements are the following:

- jQuery 1.8
- JSDoc 3
- QUnit 1.14
- Ant 1.9.4

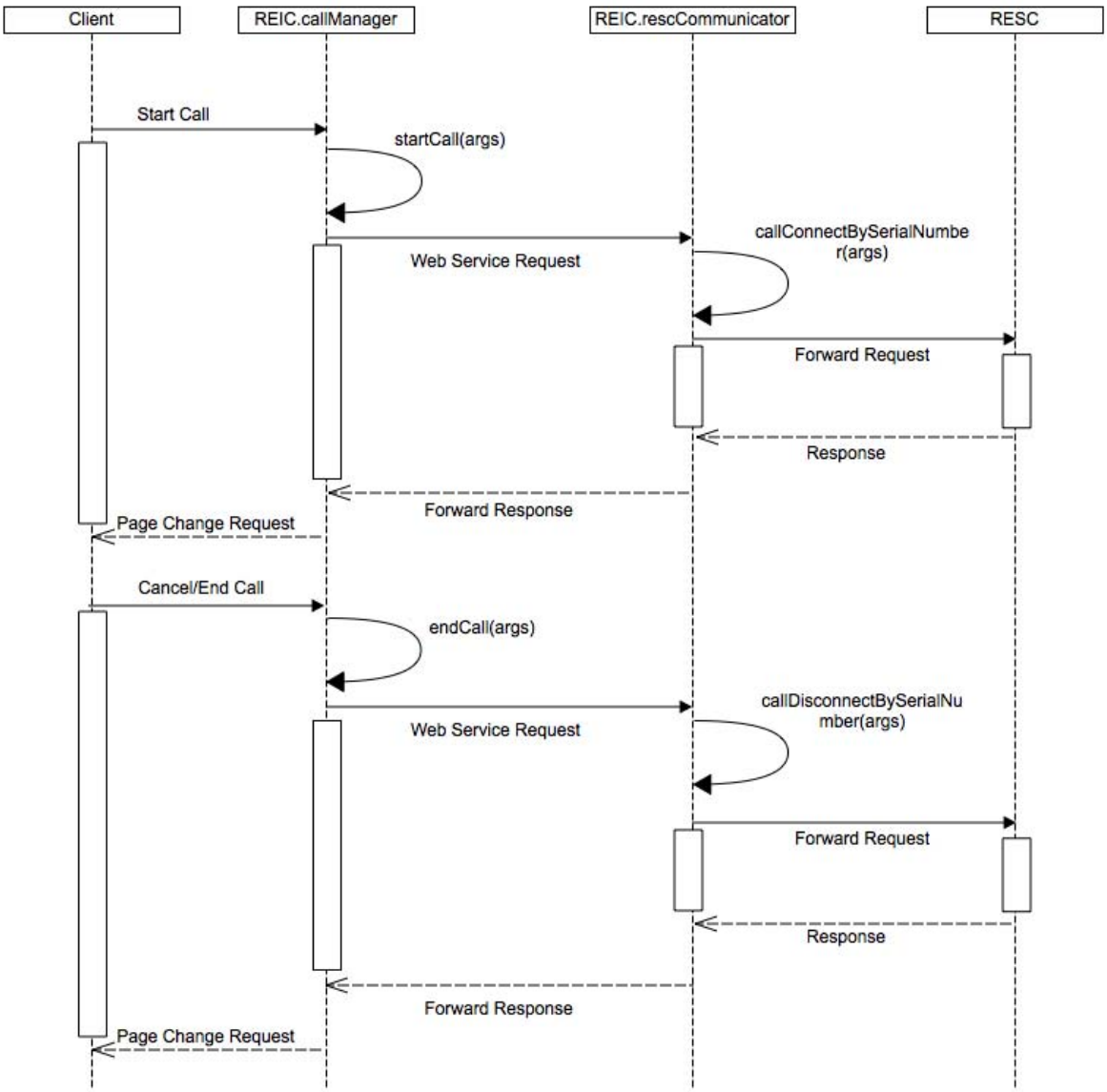
Sequence Diagrams for Components

This section contains detail communication diagrams of the following:

- Call Manager API
- Feedback Manager API
- Event Manager API

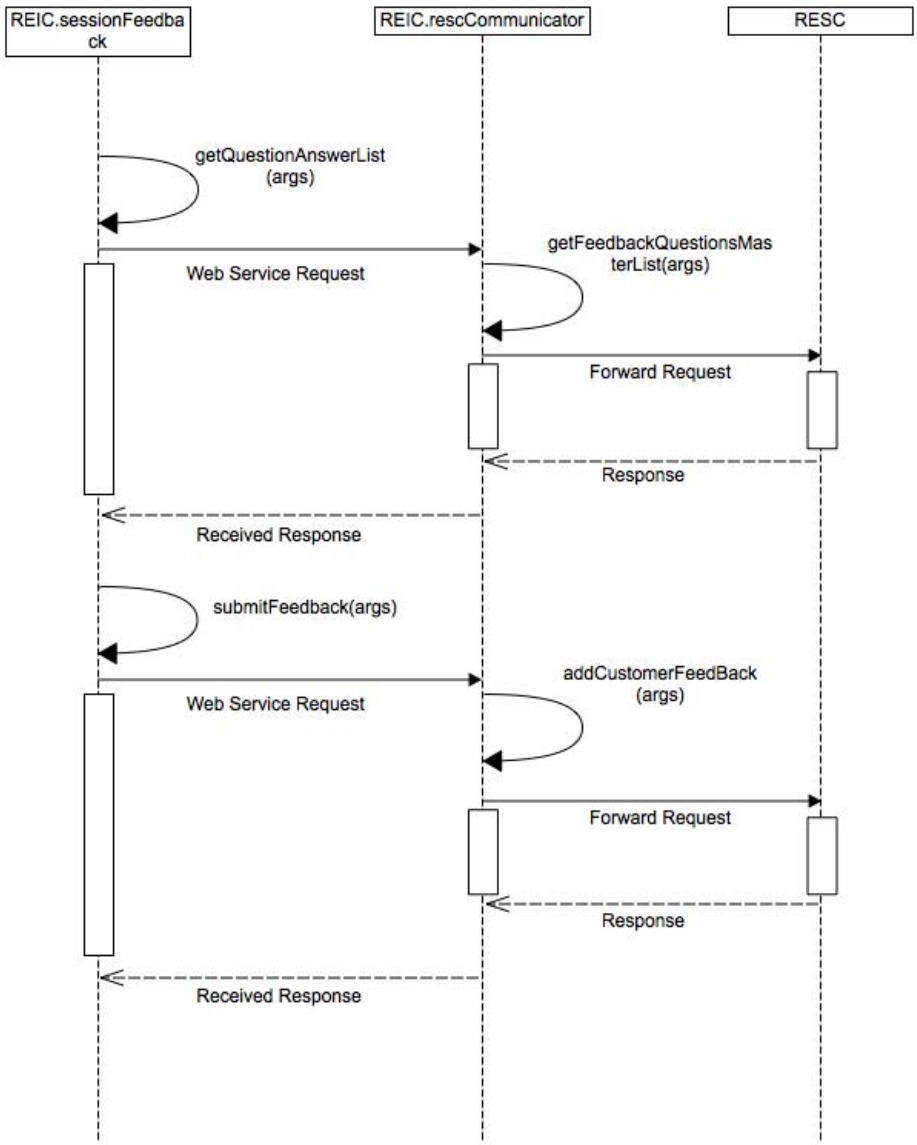
Call Manager API

Figure 2 Detailed Communication of Call Manager API



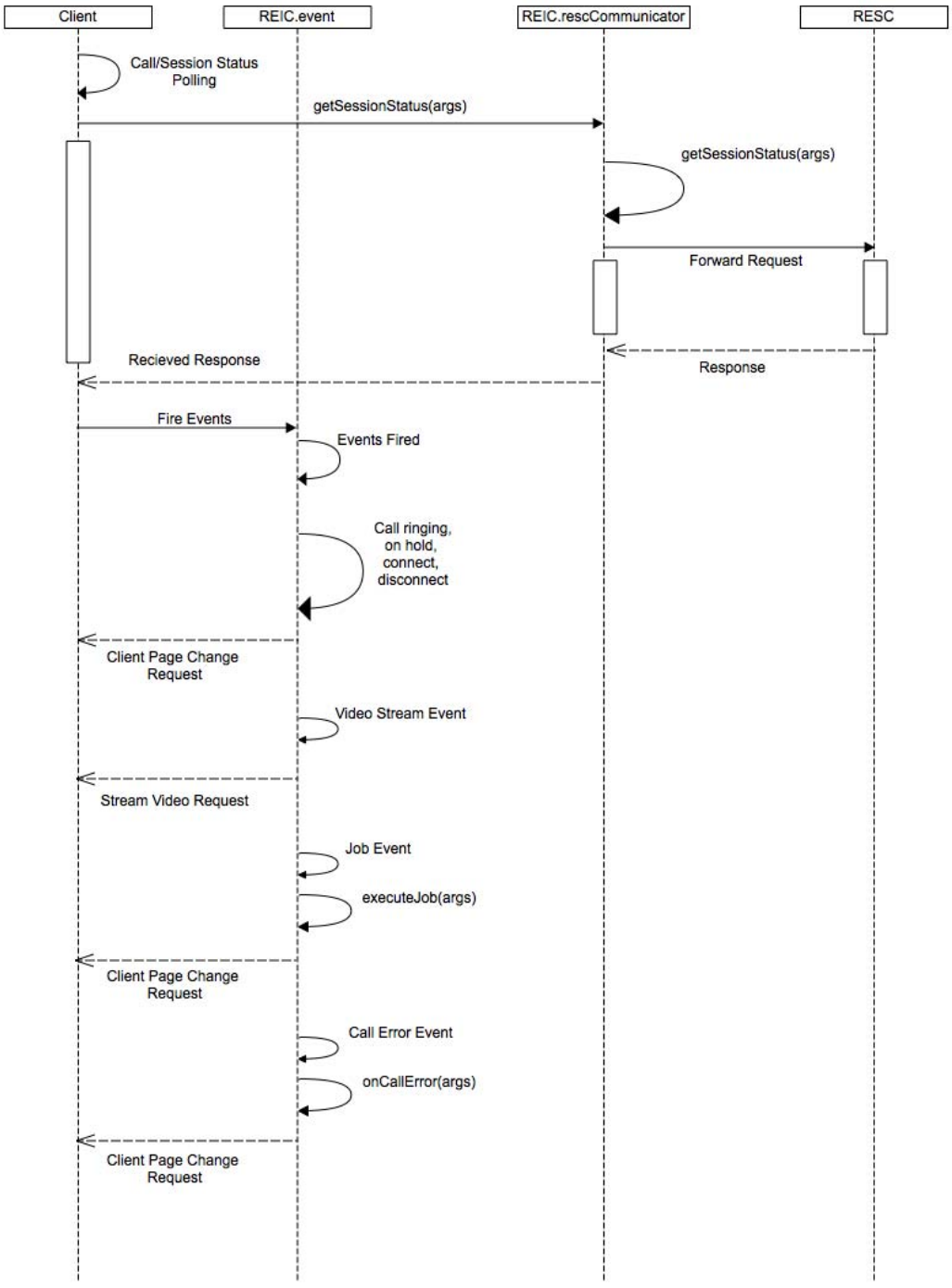
Feedback Manager API

Figure 3 Detailed Communication of Feedback Manager API in Utility Class of REIC API



Event Manager API

Figure 4 Detailed Communication of Event Manager API



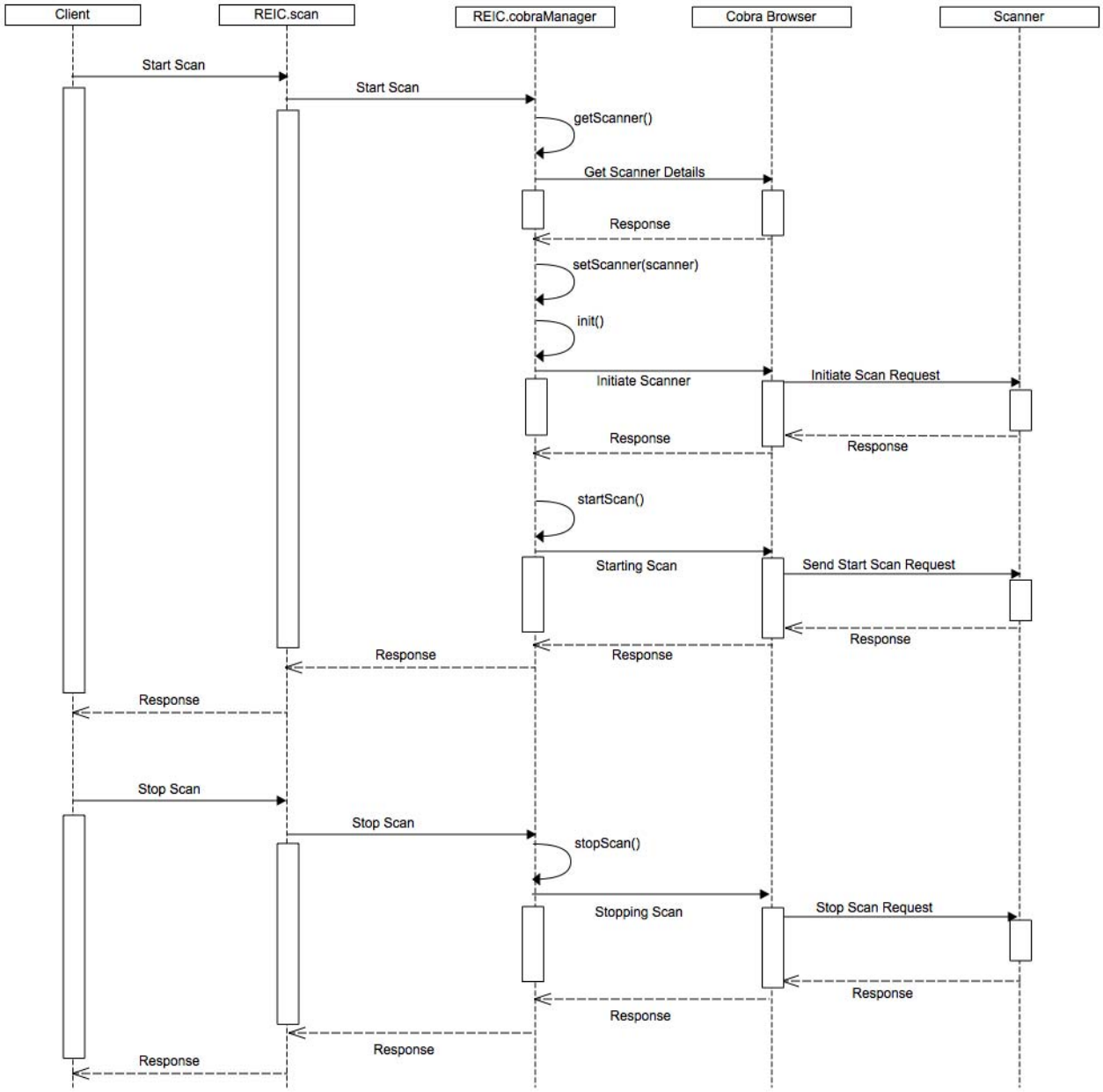
Jobs

This section contained detailed communication of the following jobs:

- Scan
- Print
- Signature Capture
- Card Reader
- VNC (Co-browsing)
- Document Camera

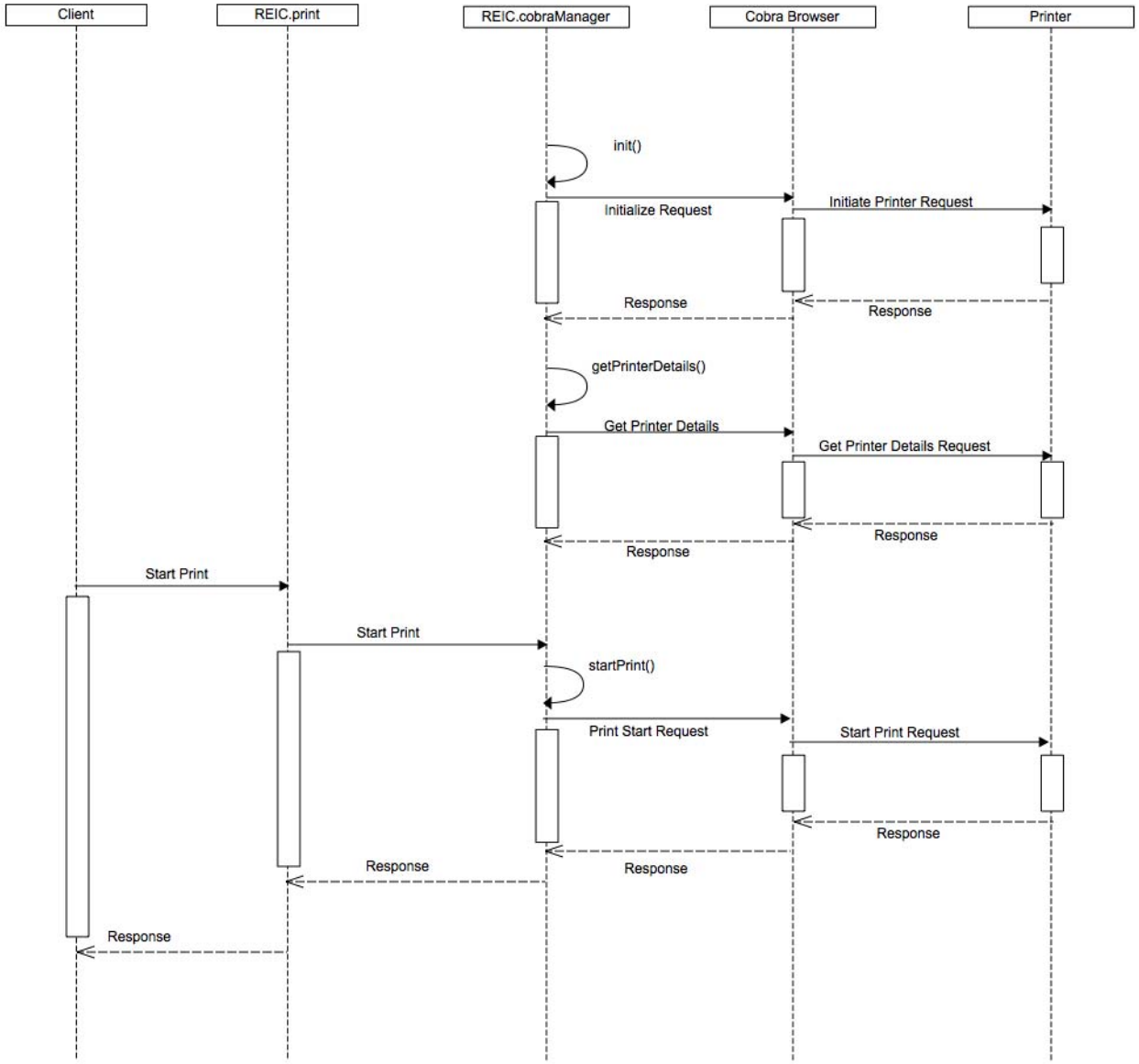
Scan

Figure 5 Detailed Communication of Scan



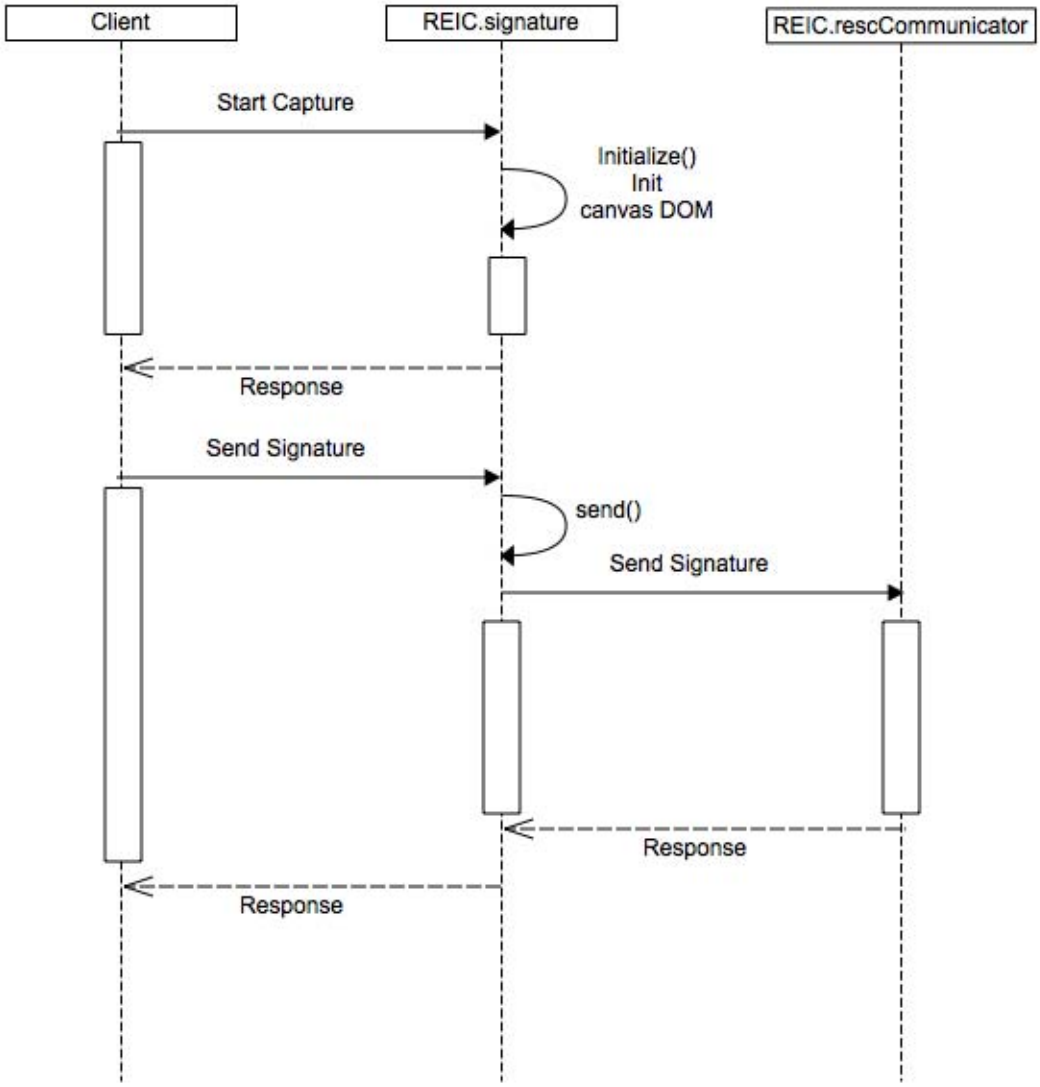
Print

Figure 6 Detailed Communication of Print



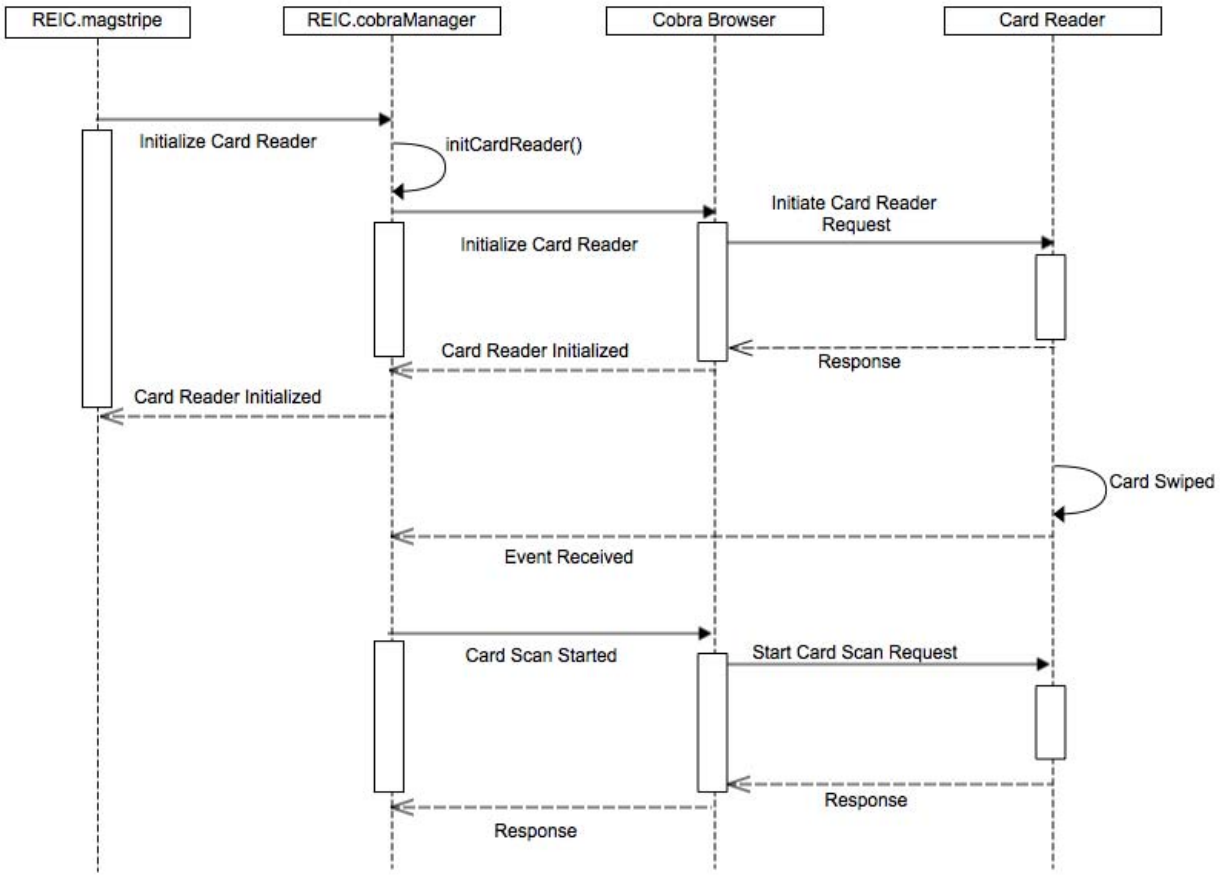
Signature Capture

Figure 7 Detailed Communication of Signature Capture



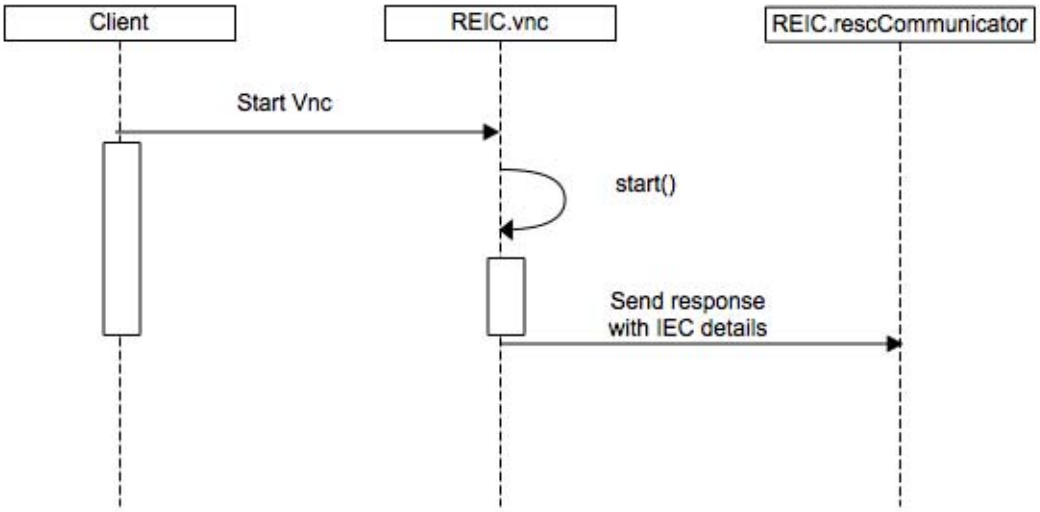
Card Reader

Figure 8 Detailed Communication of Card Reader



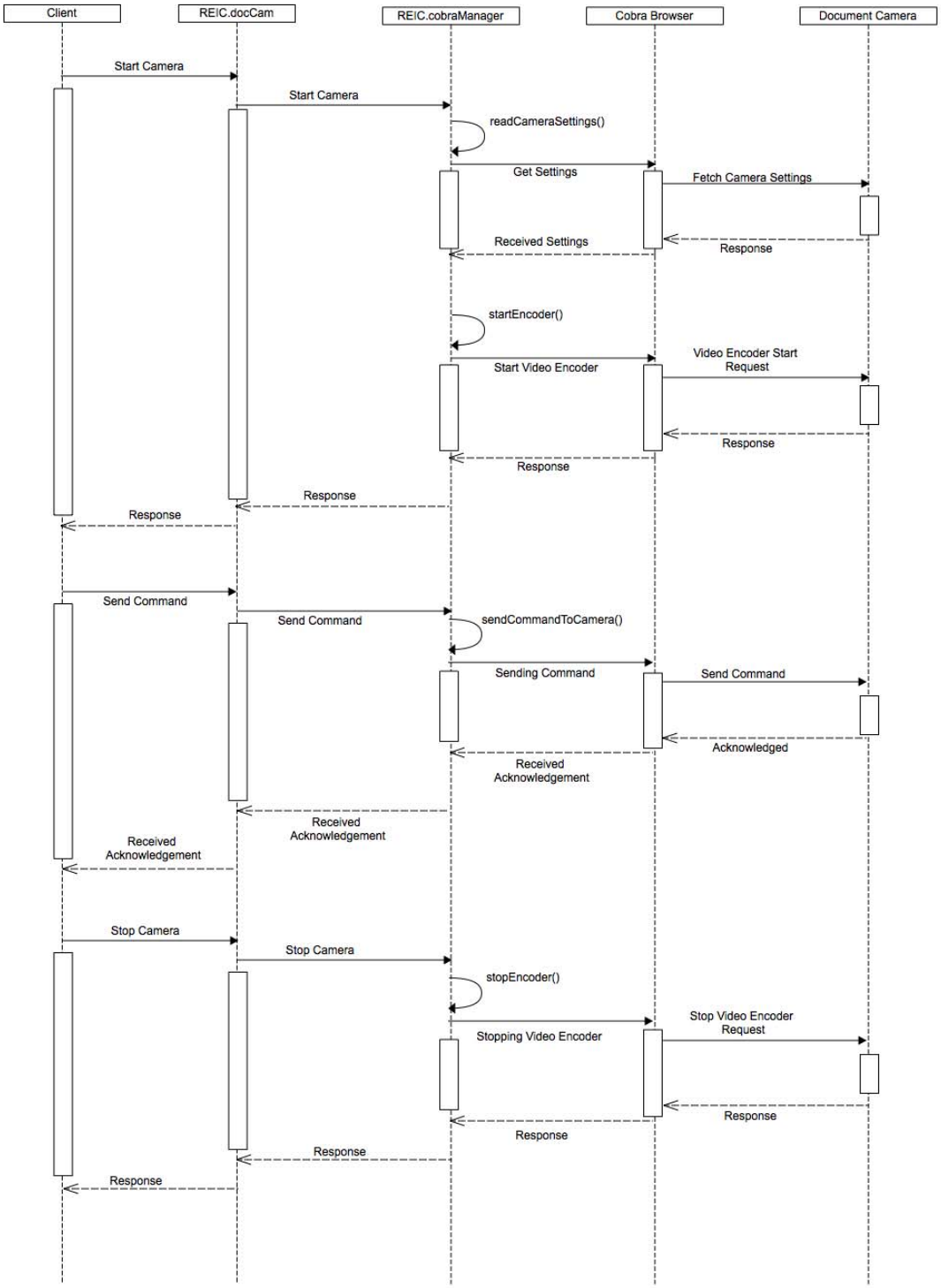
VNC (Co-browsing)

Figure 9 Detailed Communication of VNC



Document Camera

Figure 10 Detailed Communication of Document Camera



Error Details

There are two types of errors: REIC API related errors, which are thrown from REIC SDK APIs, and RESC related errors. This section contains all the error codes.

API Related Errors

The following table contains error codes that span across the APIs.

Table 3 Error Codes for APIs

API	Code	Message
Call Manager	201	Expert type is not configured
	202	System is not available
	203	Kiosk is not registered
	204	Selected Locale is not associated to any expert type
	205	Call connect error
	206	Kiosk phone is out of service
	207	There is already an active call to an expert
	208	Kiosk serial number is not configured
Print	801	Print error

Descriptions of error messages are given in the table below.

Table 4 Error Messages and Descriptions

Error Message	Description
Printer not found	Attached printer not found
The print url is invalid	The provided document URL is not valid
IEC serial is invalid	IEC serial number is not valid
Expert id must be number	Expert type identification is not a number
Expert type id is invalid	Expert type identification is empty or ‘null’
Customer id must be number	Customer identification is not a number
Session Id must be a number	Session identification is not a number
Session Id is invalid	Session identification is empty or ‘null’
Feedback must be an array of objects	Feedback provided is not an array of objects (containing a question and answer pair)
Feedback array must contain objects	Feedback array does not contain a plain object
Locale code is invalid	Locale code is empty or ‘null’ for feedback
Job Id must be a number	Job identification is not a number

Error Message	Description
Job id missing	Job identification is missing
Scanner is busy	Scanner is busy scanning
Scan operation was cancelled	Scan is canceled due to an internal error
Scan invalid argument	Invalid scanner name is passed
Scanner is not supported	Scanner is not supported
Scanner not found	No attached scanner
Invalid image format	Invalid image format supplied
Invalid image data	No image data is provided
Invalid scanner name	Scanner name is empty or ‘null’
Invalid scanner source name	Scanner source name is empty or ‘null’
Command Id must be a number	Command is not a number
Command id is missing	Command identification is empty or ‘null’
Valid card reader is not found	No attached card reader
Invalid card data	Card data after swiped is invalid
Invalid signature pad	Signature pad (canvas DOM) is empty or ‘null’ was provided for signature
Invalid baud rate	Baud rate is not a string, empty, or ‘null’
Invalid data bits	Data bit is not a string, empty, or ‘null’
Invalid flow control	Flow control data is not a string, empty, or ‘null’
Invalid parity check	Parity data is not a string, empty, or ‘null’
Invalid stop bits	Stop bit data is not a string, empty, or ‘null’
Invalid vnc host	VNC server IP is invalid
Invalid vnc port	VNC port is invalid

RESC Related Errors

The following table contains the error codes and messages related to RESC errors.

Table 5 Error Codes Related to RESC Errors

Code	Message
101	System error
110	Illegal argument
111	Record already exists
114	Record not found
115	No active REM session

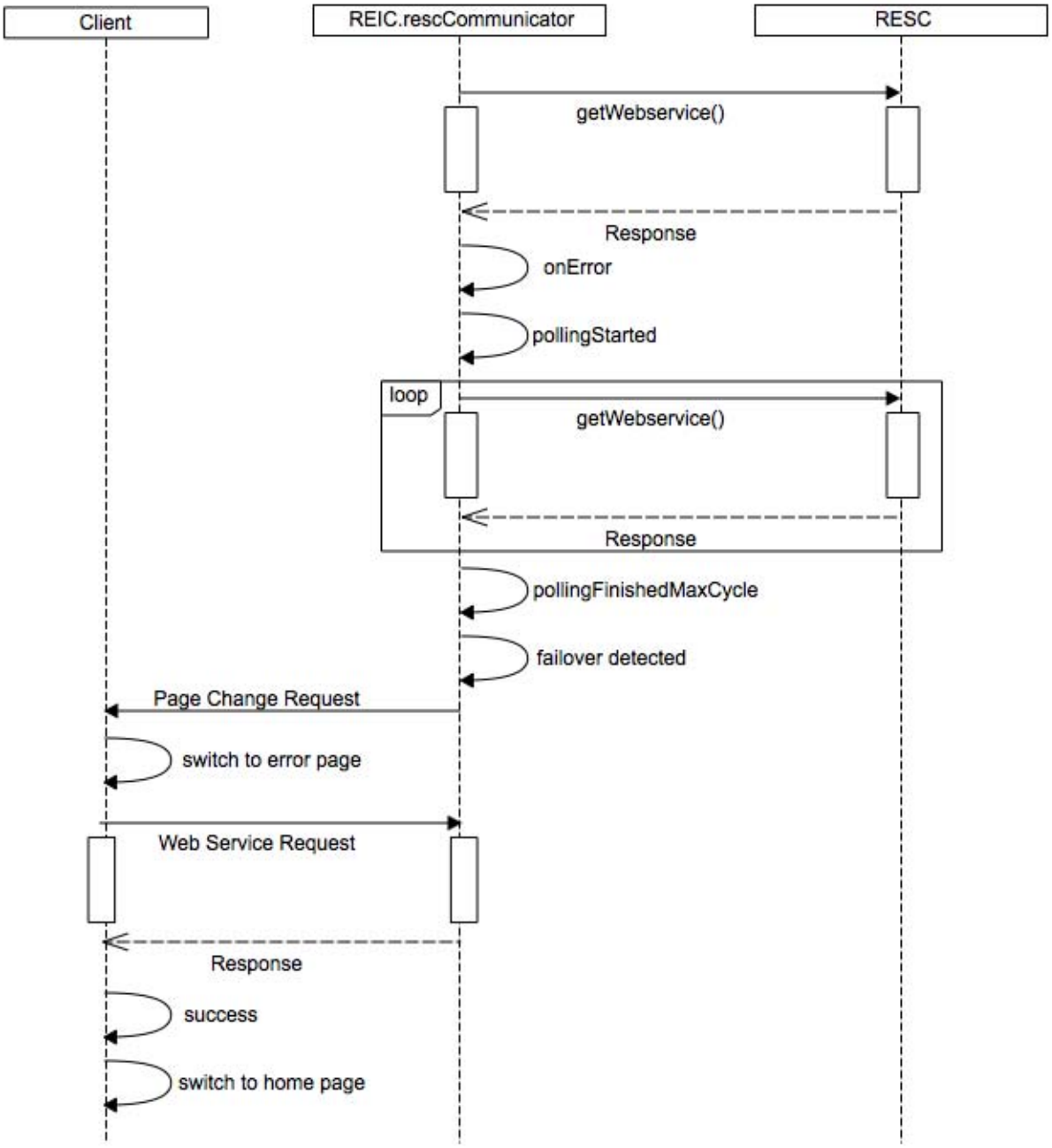
Fail Over

When a call to a web service failure is detected, polling is started from the REIC to check the web service status periodically for a configured number of times (‘`server.down.faildetect`’) in the REIC properties. If web service becomes active, polling is stopped; otherwise, if the polling cycle is complete and still web services response is in error, a failover is detected (both nodes are down). REIC sends a request to the client UI to change its current page to an error page. Another polling is started from the error page once it is loaded, to check for an active admin service. When any one of the nodes becomes active, the error page moves to the home screen.

Polling from an error page and transition from an error page to the home page are not handled by the REIC SDK. The client application needs to implement this feature.

A high-level sequence diagram is presented below for failover scenarios:

Figure 11 Sequence Diagram for Failover Scenarios



Adding the REIC SDK to a Client Application

The following is needed to add the REIC SDK to a client application:

Step 1 Copy the **reic** and **reic-SDK** folders to the client content server using the directory structure shown in the figure below.

Figure 12 Directory Structure

```
App -> reic [Folder]
App -> reic-SDK [Folder]
App -> index.html
App -> Other Client side files
```

Step 2 Include the REIC-SDK library (shown in the figure below) in the client application.

Figure 13 REIC-SDK Library

Package	Code
REIC-SDk	<script type="text/javascript" src="reic-SDK/src/Main.js"></script> <script type="text/javascript" src="reic-SDK/src/CobraManager.js"></script> <script type="text/javascript" src="reic-SDK/src/Config.js"></script> <script type="text/javascript" src="reic-SDK/src/Log.js"></script> <script type="text/javascript" src="reic-SDK/src/Common.js"></script> <script type="text/javascript" src="reic-SDK/src/TaskManager.js"></script> <script type="text/javascript" src="reic-SDK/src/CallManager.js"></script> <script type="text/javascript" src="reic-SDK/src/RescCommunicator.js"></script> <script type="text/javascript" src="reic-SDK/src/SessionFeedback.js"></script> <script type="text/javascript" src="reic-SDK/src/Printer.js"></script> <script type="text/javascript" src="reic-SDK/src/Scanner.js"></script> <script type="text/javascript" src="reic-SDK/src/Signature.js"></script> <script type="text/javascript" src="reic-SDK/src/MagStripe.js"></script> <script type="text/javascript" src="reic-SDK/src/Vnc.js"></script> <script type="text/javascript" src="reic/js/remoteexpert.js"></script> <script type="text/javascript" src="reic/js/sipBroadCast.js"></script> <script type="text/javascript" src="reic/js/reic.js"></script>
jquery	<script type="text/javascript" src="lib/jquery/jquery-1.11.1.min.js"></script>