Appendices

- Appendix A – Integration with symlink use case
- Appendix B – Integration with third party application or HTTP service
- Appendix C – Integration with plugin use case
- Appendix D – Chromium Embedded Framework (CEF), WebView and Electron
- Appendix E – Integration with an external application

Appendix A – Integration with symlink use case

Actors:

User, User Portal, Spark Gateway

Preconditions:

User Portal:

- Have user credentials in plain text.

Spark Gateway:

- Configure password in gateway.conf.
- Allow IP addresses of User Portal to access the Spark Gateway API (Optional, Admin Manual 3.25).
- Create RDP servers with config.html or HTTP API (Integration Guide 4.1), and make it as a white list.
- Allow symlink access only by setting symlinkOnly = true in gateway.conf. Server id or address will not be allowed to create a connection.
- Disable VNC access by setting vnc = false in gateway.conf (SSH, Telnet are disabled by default).

Basic flow:

1. User login to User Portal.
2. (optional) User Portal create a server on Spark Gateway with HTTP API if that server is not created yet:
   http://gatewayAddress/SERVER?id=serverId&displayName=Name&server=hostName&gatewayPwd=passwordInGateway.conf&...; Check if server exists:
   http://wthink/SERVER?action=list&gatewayPwd=21232f297a57a5a743894a0e4a801fc3&id=serverId;
   The gateway will return HTTP Status code 500 and {"error:": "not found"} in JSON format.
3. User Portal create a symlink on Spark Gateway with HTTP API:
   'http://gatewayAddress/SYMLINK?symlink=symlinkId&server=existingServerId&validTime=8h&gatewayPwd=passwordInGateway.conf&parameters=' + encodeURIComponent('user=domainUser&pwd=domainPassword&domain=domain');
4. User portal construct a connection link and display it to user:
   http://gateway/rdpdirect.html?symlink=symlinkId&displayName=nameOnBrowserTitle; or

use the Spark View JS library directly to create a connection:
`var rdp = new svGlobal.Rdp('wss://gateway/RDP?symlink=symlinkId&..', width, height, color);`

5. User click the link and connect.
6. User portal delete the symlink when user exist or close the browser: http://gatewayAddress/SYMLINK?symlink=symlinkId&action=delete

Pros:

- Easy and secure.
- Symlink will be invalid or deleted and it cannot be reused by other.
- No need to send the user credential to the browser side.

Cons:

- User domain credentials need to be sent to the gateway in plain text.

Appendix B – Integration with third party application or HTTP service

Actors:

User Portal, Spark Gateway, Third party application or HTTP server

Preconditions:

User Portal:

- Prepare a token which can be used to verify user.

Spark Gateway:

- Configure authToken.name, authToken.exec, authToken.sucessCode in gateway.conf.

Basic flow:

1. User Portal send the token to Spark Gateway along with other parameters.
2. Spark Gateway check if the token name is same as the value configured in authToken.name. It’s same, then.
3. Spark Gateway execute the application or HTTP Request(GET) configured in authToken.exec, if the return code or HTTP Status code is same as the authToken.sucessCode, gateway will allow the connection, otherwise reject it.

Appendix C – Integration with plugin use case

Actors:

User, User Portal, Spark Gateway

Preconditions:

User Portal:

- Have user credentials in plain text.
- Provide a service to verify the user (optional).

SparkView:

- Write a simple plugin in java for Spark Gateway.
- Disable VNC access by setting vnc = false in gateway.conf (SSH, Telnet are disabled by default).

Basic flow:

1. User login to User Portal.
2. User Portal encrypt the user credential and other information like User Portal session id as a token string, for example: 'user=domainUser&pwd=domainPassword&domain=domain&session=id'.
3. User Portal construct a connection link and display it to user: http://gateway/rdpdirect.html?token=myEcryptedToken&displayName=nameOnBrowserTitle&otherParameters; or use the Spark View JS library directly to create a connection: `var rdp = new svGlobal.Rdp('wss://gateway/RDP?token=myEycryptedToken&..', width, height, color);`
4. User click the link and connect.
5. SparkView plugin decrypt the token, verify the user session (optional) and put the decrypted RDP parameters back (Please check the plugin example for details: http://www.remotespark.com/Plugin.zip)

Pros:

- User credentials are encrypted.
- Encryption and decryption are done on server side which is pretty safe.
- Plugin can also be used to verify if user is form a valid session, or extend the SparkView functionality.

Cons:

- Need to write some lines of Java code for Spark View gateway.
- You may need to recompile the plugin when upgrading to a new version of Spark Gateway (if the plugin interface changed).
- Encrypted user credentials still need to be sent to the browser.

Both use cases are recommended. You can also consider to use them together (symlink with plugin), so user credentials can be encrypted and don’t need to be sent to the client side.

If User Portal cannot have the user credentials in plain text, you can consider to create a temporary windows user account for every user and remove this account later.

Appendix D – Chromium Embedded Framework (CEF), WebView and Electron

You can use CEF or Electron to make a standalone client, so SparkView client can access local resource directly (clipboard), and override some shortcuts keys reserved by the browser (Ctrl+T/W etc).

1. Let Spark View know it can access the clipboard directly by setting `directClipAccess: true` in appcfg.js.
2. For CEF, you need to enable cef_state_t javascript_access_clipboard.
3. For WebView, please check: https://stackoverflow.com/questions/4200259/tapping-formfield-in-webview-does-not-show-soft-keyboard
4. For Electron, you can also use `mainWindow.webContents.executeJavaScript('hi5.appcfg.directClipAccess=true')` to inject the code.

Appendix E – Integration with an external application

In addition to the known and internal verification options, an external application can also be used for verification. This is usually easier than using your own plugin.
To do this, add the following lines to gateway.conf

```
authToken.name = myToken
authToken.exec = C://MyApps//auth.exe %1
```

SparkView will replace `%1` with the value of the token in the WebSocket URL. The application can then verify or decrypt this value. Additional parameters can also be written back to standard output. These have the following format:

```
__SG_ARGS=true\tArg1=v1\tArg2=v2
```

The output must start with `__SG_ARGS=true\t`, where the different arguments are then separated by the tab character (`\t`). SparkView will then add these parameters back to the WebSocket URL.