Vrije Universiteit Brussel

FACULTY OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE AND APPLIED COMPUTER SCIENCE

Advanced Linking and Annotations for Multimedia Documents and Applications

Bachelor Paper

Tom Strickx

Promotor: Beat Signer

Advisors: Ahmed A. O. Tayeh

1 Abstract

In this report we will document the implementation of four plug-ins for the link service application, which offers advanced linking capabilities, beyond the limited restrictions of the supported document formats. We will explain the general architecture of the application, as well as a clear overview on how to implement a new plug-in for an external application. The issues we encountered while we worked on these plug-ins will aide us in providing several suggested enhancements for the link service.

2 Introduction

A document is not a standalone object, but exists in relation to other objects. Unfortunately, current hypermedia solutions only support simple forms of linking, for example, many document formats allow linking to entire third-party documents, but most do not have a finer grained linking system, which allows linking to a specific section of a document (e.g. XLink). This is the basic unidirectional link model we all know, and was instrumental to the success of the World Wide Web. In the unidirectional link model, a document that is linked has no knowledge of the document it is being linked from. The advent of the Extensible Markup Language (XML), and its link model (XLink) (DeRose, Maler, Orchard, and Trafford (2000))led to a major step forward towards advanced linking mechanisms, beyond unidirectional links. Unfortunately, XLink only deals with XML documents, and does not support any other non-XML document models or formats. Therefore a new and extensible link model and architecture should be used, which supports more advanced linking features, but also supports multiple existing and emerging formats. A metamodel, the resource-selector-link (RSL) model (Signer and Norrie (2007)), was proposed that would support these features. This resulted in the open cross-document link service.

2.1 Background

By using an external link service which defines, stores and manages the general hyperlink concepts, one can overcome several limitations with other advanced linking mechanisms, such as XLink, where the assumption is made that all linked objects have a tree-like structure, which is unfortunately not always the case, for example the PDF standard. By keeping the definition of sources, targets and resources abstract, the actual implementation can be done using a plug-in mechanism. This plug-in mechanism also allows the link service to be extensible to support existing as well as emerging new document formats.

3 Cross-Document Linking Architecture

The general architecture of the link service is illustrated in Fig. 1. The central component in the service is the core link service, which is based on the RSL metamodel we previously mentioned. The core itself is an extensible system, which supports arbitrary document formats. This is achieved by offering permission to all other modules to import the core RSL package. This allows new data plug-ins to extend the basic RSL resource and selector, as well as the visualisation component. A specific document format extension for the RSL resource and selector is called a data plug-in. For each document format that has to be rendered in the application browser, a visualisation plug-in has to be implemented as well, which renders the specific document, as well as visualise any selectors that have been defined. It has to be noted that most document formats have their own, usually proprietary, third-party applications. For these applications, a visual plug-in should also be provided, however these plug-ins do not directly support the CRUD operations, but communicates with the link browser component to exchange information. In the current implementation, a third-party application can communicate with the link service using either WebSockets, raw TCP sockets, or a REST interface. The link service itself is implemented using the Open Service Gateway Initiative (OSGI)

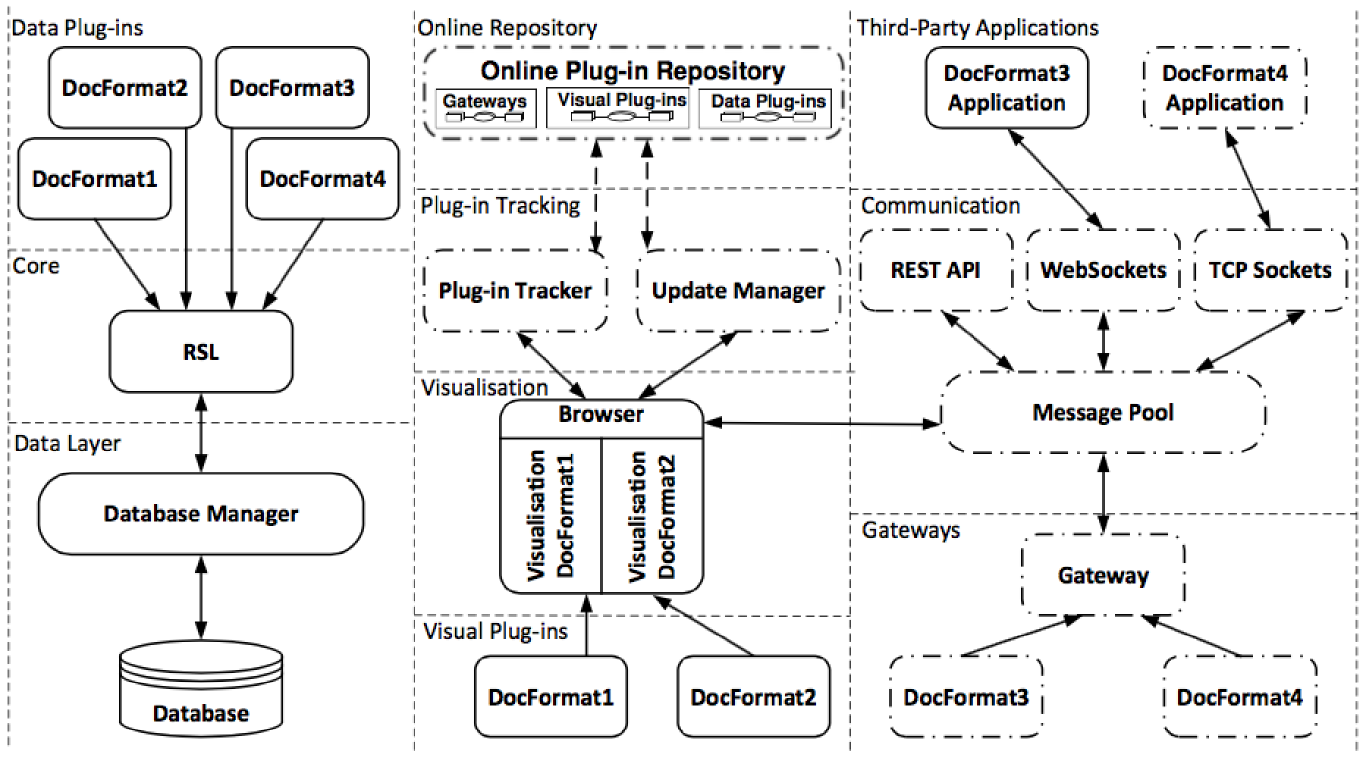

Figure 1: General open cross-document link service architecture (Tayeh and Signer (2015))

specification. The OSGI specification defines a dynamic modular system for Java. Various server applications make use of OSGI (e.g. IBM Web-sphere), and the Eclipse IDE also uses OSGI to enable modularisation of its components, and to support dynamic extensions via various plug-ins. The OSGI system offers a better mechanism for code sharing compared to Java JAR files, which share arbitrary code, by explicitly defining export packages, as well as import packages to be used. By this clear definition of packages, other installed modules cannot misuse any code of other modules. Since the link service should also be dynamically extensible to support new document formats, without a need for redeployment, the OSGI system can be used to provide dynamic extensibility, which allows users to download plug-ins for new document formats on demand, without a need for recompilation.

4 Plug-ins

The current iteration of the project already has 5 existing plug-ins, either as a visual plug-in (text and pdf), or as an external plug-in (HTML). As a contribution to this project, we intend to extend the plug-in availability by adding 4 more plug-ins: Microsoft Word, Microsoft Powerpoint, Adobe Acrobat, and Youtube. The implementation of three was finished successfully, while the implementation of the Adobe Acrobat plug-in encountered several issues which were not resolvable within a reasonable timeframe. In this section we will give a general overview on the plug-in development, as well as the basic requirements a new plug-in has to fulfil, and how an external application plug-in can communicate with the link service.

4.1 Loading plug-ins

As previously mentioned, the link service is implemented using the OSGI framework, allowing for dynamic loading of plug-ins (or bundles in OSGI terminology). In the link service its case, OSGI provides several bundle related events, as well as appropriate event listeners. If a plug-in is properly identified as either a data plug-in or a gateway, it will be added to the its appropriate hash table, associating the plug-in with the mime type it handles.

4.2 Writing new plug-ins for external applications

If we want to extend the link service with a new document format, two OSGI plug-ins have to be created: a data plug-in, which extends on the existing RSL resource and selector, and an application gateway. First we will discuss the data plug-in. Since the RSL resource and selector concepts are abstract, it is up to the programmer to choose a representation of both a resource and a selector, and is only limited by the required functionality of the classes. For a resource, this functionality is very basic, and only requires a constructor, which sets the mime-type of the application, and the

URI of the document location. The mime-type is used to differentiate communication between different external plug-ins. The selector also requires a new constructor, as well as a setter and getter for the selector itself. The selector can be anything the programmer wants, for example an xpointer for an XML document, or a time reference for a multimedia document. The link service gets additional information about the data plug-in through the manifest file, which contains details about the plug-in. A data plug-in's manifest file should contain the following attributes:

- Extension-Name
- Extension-Mime
- Extension-Type

Once a data plug-in has been defined, it can be used by the application gateway. The gateway is responsible for interpreting the communication between the core link service, and the third-party application. The gateway project should also define some custom properties in its manifest file:

- Extension-Name
- Extension-Mime
- Extension-Type
- Extension-Class

These properties allow the main application to know which class to load as gateway, as well as which gateway to hand over the communication data from an external application. The gateway itself is an implementation of the abstract `GateWay` class. A number of member functions need to be implemented to make the gateway functional:

- getMime
- launchApp

- getURIofOpenedResource
- getResource
- getResourceId
- openDocument
- getTargetEntityID
- updateView
- deseraliseSelections
- getIdofEntityToUpdateOrDelete
- updateSelector
- updateResource

Some of these functions are trivial, such as the `getMime` function, which should simply return the mime type for which the gateway is responsible, and should match the Extension-Mime property in the manifest file. The most important functions are `updateView` and `openDocument`, which will be discussed at length for the implemented plug-ins.

4.3 Communication with the Cross-Document Linking Service

The link service currently supports several methods of communication for external applications:

- Websockets
- Raw TCP sockets
- REST

To start communication with the link service, the external application, once connected to the link service using one of the above protocols, has to identify itself by sending the mime-type for which it will handle the user interface. Once the mime-type message has been received by the link service, it will associate the communication session with the mime type, so all events for this mime-type will be sent to this session. This currently means that two different applications cannot communicate if they identify themselves using the same mime-type. The messages between the service and the external application have to be in the JSON format. The message handlers for the different protocols are responsible for the initial handling of the message, before handing it over to the responsible gateway. This is done by using the c̈ommandättribute. Depending on the given command, the appropriate function in the gateway will be called. The currently supported commands are:

- showTarget: This command asks the link service to open a given entity (either a resource or a selector), with a specified mime-type
- updateEntity: Tell the link service to update an entity with a given id. This is used to modify links
- deleteEntity: Tell the link service to remove a given entity
- isOpen: Inform the link service a given resource has been opened. This will result in a message sent back containing the available selectors in the resource

Depending on the command, the content of the rest of the JSON object is variable.

5 Microsoft Office Plug-in

We will implement a plug-in for Microsoft Word and Powerpoint, which will tie-in with the cross-document linking service using the websocket communication channel. Currently, Microsoft offers two ways to add additional functionality to their range of Office applications: Either one uses the well-established `C#` and .NET API, or use the fairly recent Javascript API. The Javascript API has the major advantage it is also supported on other platforms, such as Office for iPad, and Office 365. The choice was made to implement the plug-in using the Javascript API, as this allows for a broader reach of the plug-in, without having to port it to different platforms.

5.1 Office SDK

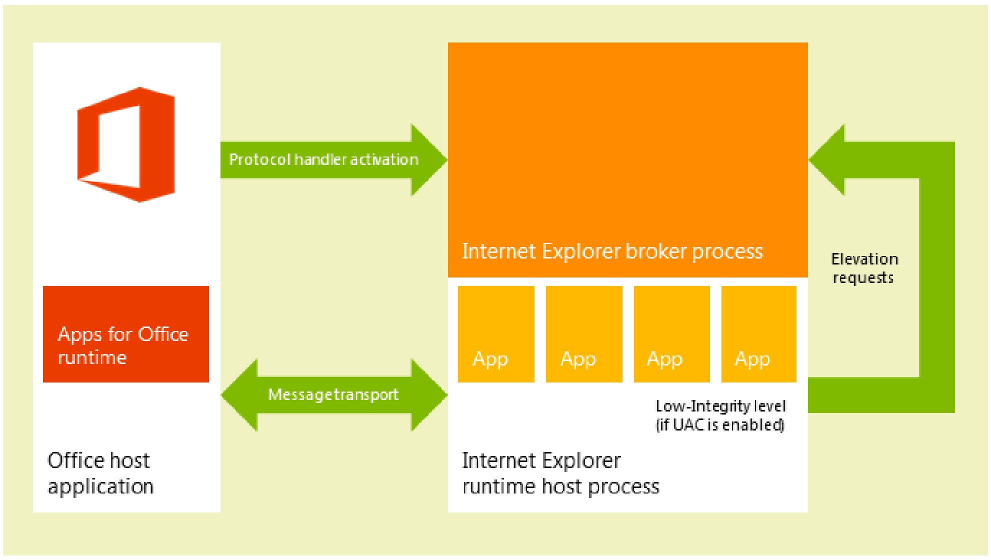
Protocol handler activation
Internet Explorer broker process
Elevation requests
Apps for Office runtime
App
App
App
App
Messagetransport
Low-Integrity level (if UAC is enabled)
Office host application
Internet Explorer runtime host process

Figure 2: Office Add-in Runtime (Microsoft (n.d.)

Microsoft provides an extensive Javascript API to build apps for Office. The extension itself can either be a task pane application, residing in the sidebar, or a content application, where it resides in the document itself. Since a content application would be very obtrusive for our use-case, it was opted to implement a task pane. The application itself is ran in a sandboxed Internet Explorer instance, with limited access to the host Office application, as can be seen in Fig. 2. This offers significant improvements to the users their privacy, as well as the security of the application, as well as make apps easy to install and uninstall, since no dynamic link libraries (.dll) or executables are needed.

The basic components of an app for Office are an XML manifest file and the default webpage of the app. The manifest defines various settings including the URL of the webpage that implements the app's UI and custom logic. The Javascript API provides an Office namespace, which contains objects that are used to interact with content in an Office documents, such as getting the current selection made with the cursor.

5.2 Microsoft Word

Microsoft Word Add-in

Since the Add-in for Office runtime provides a full-fledged browser environment, we could make use of HTML5, CSS3, Javascript and websockets. This provided an excellent foundation on which the plug-in could be built. The add-in itself is written using a combination of HTML and CSS (to provide the user interface in the task pane), and Javascript (custom logic and communication with the link service). The provided Javascript functions rely heavily on the asynchronous programming paradigm, where a callback is called once the function returns.
The initial user interface contains 3 buttons, as can be seen in Fig. 3. The explicit connection to the link service is required, since the API will re-instantiate with every opened document, which would result in multiple websocket connections to the link service with the same mime-type otherwise. A connection to the websocket server is made using the available WebSocket object in Javascript. This is done in the `connectWebSocket` function, which will also define an `onopen` event handler, as well as an `onmessage` event handler. The `onopen` handler will send two initial commands to the link service, one to identify itself with its mime-type, and the other to retrieve any available selectors in the document using the `isOpen` command. Once a connection is made, the user can access the available selectors, and open, update or delete them. On every received (and sent) message, the `onmessage` event handler is called.

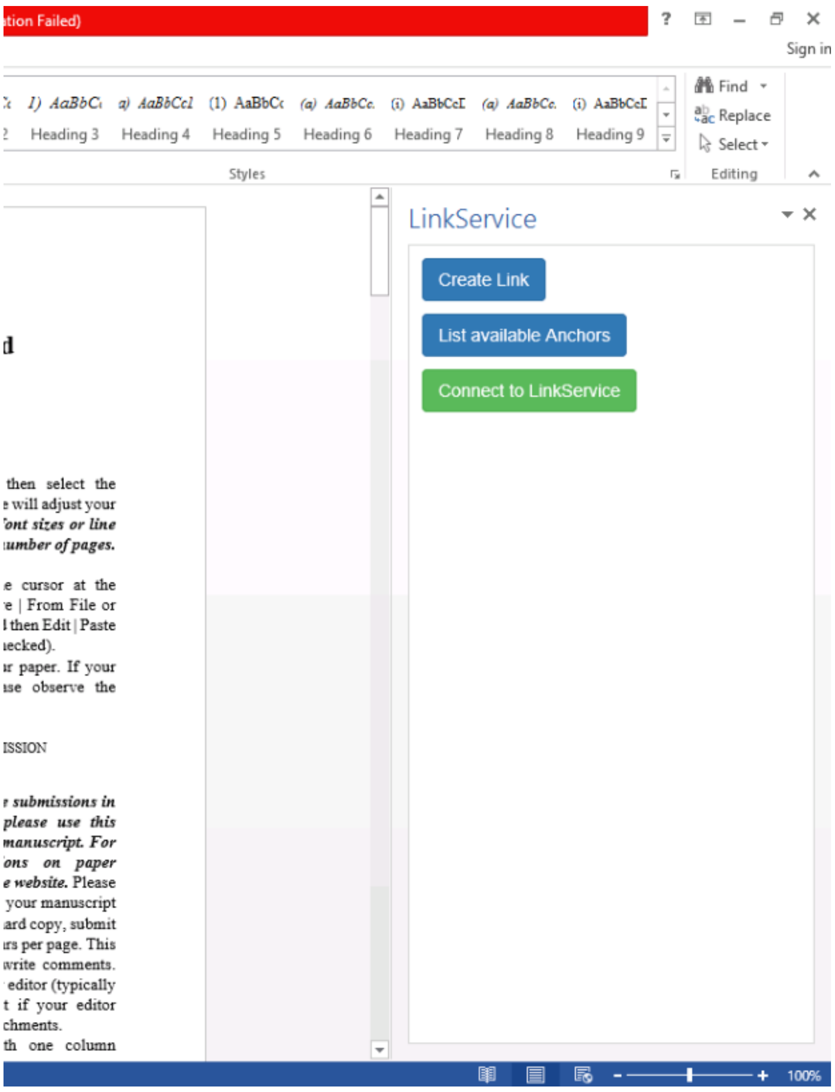

Figure 3: Microsoft Word Add-in user interface

```
function connectWebSocket() {
        var host = "ws://localhost:8025/websockets/path";
        socket = new WebSocket(host);
        socket.onopen = function (openEvent) {
            document.getElementById("message").innerHTML =
               'WebSocket Status:: Socket Open';
            socket.send(JSON.stringify({ "mime": "text/word" }));
            var getAnchors = { "command": "isOpen",
                "uri": encodeURIComponent(Office.context.document.url) };
            socket.send(JSON.stringify(getAnchors));
        };
        socket.onmessage = function (messageEvent) {
            console.log("Websocket message received");
            parseMessage(messageEvent.data);
        }
        $("#connectService").remove();
        var submitButton = $('<button />',
                        { text: "Submit Links",
                        click: function () { createLink(); } });
        $("#UI").append(submitButton);
    }
```

Listing 1: connectWebSocket Javascript function

```
function parseMessage(jsonObject) {
        var jsObject = JSON.parse(jsonObject);
            if ('command' in jsObject) {
                        parseCommands(jsObject);
            }
}
```

Listing 2: parseMessage: offer abstraction in case of added object complexity

```
if ('Resource' in jsonObject) {
        if (encodeURIComponent(Office.context.document.url) === jsonObject.Resource.uri) {
                if ("Anchors" in jsonObject) {
                        for (var i = 0; i < jsonObject.Anchors.length; i++) {
                                idToBinding[jsonObject.Anchors[i].Selector.xpointer] =
  ↪   jsonObject.Anchors[i].Selector.id;
                                idtoTarget[jsonObject.Anchors[i].Selector.xpointer] =
  ↪   jsonObject.Anchors[i].Targets;
                            }
                            if ("Highlight" in jsonObject) {
                                gotoBinding(jsonObject.Highlight.xpointer);
                            }
                }
        }
         else {
                write("Please open the file: " +
  ↪   decodeURIComponent(jsonObject.Resource.uri));
        }
}
```

Listing 3: Extract of parseOpen, showing the mapping of xpointers to database IDs

The handler will proceed to call the `parseMessage` function, using the event data as parameter. In `parseMessage`, the received string is parsed into a Javascript object, after which it is checked for specific commands. The `open` command is the most import one, since it is both used to open specific selectors within the document, as well as provide the document with all available selectors and their attributes (such as target and id). If a `Highlight` property is provided, the add-in will call `gotoBinding` with the binding id. This will make the cursor go to the targeted text, as well as scroll the document to the appropriate location. To identify selectors in a Word document, we opted for using the provided Office bindings. A binding can be used to reference a bound text selection within a document (it can also be used to reference tables in Excel, as well as Word). The `addBindingFromSelection` function will create a new selector in the document from the currently selected text. In the background this will create a binding in the document, and add a JSON object containing the unique identifier of the binding to a linkqueue. These bindings are custom content controls, which are embedded using

Open XML parts within the document itself, which means that, for a document to be linkable, it has to be saved after the creation of the selectors.
Since a major use-case of the link service is multi-directional links with many-to-many properties, the user can create multiple selectors before submitting them to the link service. This is achieved by adding the selector objects, containing their binding id, to a linkqueue array. This linkqueue array will be added to a Javascript object containing the resource information once the user wants to submit the links.
Once a user chooses to open an available link, `openLink` is called with the binding identifier. As the link service has no concept of the binding identifier, this has to be translated to an integer id. This is done using a Javascript object which maps the binding identifier to the link service ids (multiple ids are possible, as a link can point to multiple targets). After the user selects the appropriate link (only in case of multiple targets), the `showTarget` command is sent to the link service, together with the id and mime-type of the target link. The link service will then handle this, and delegate it to the appropriate plug-in. The update and delete functionality follow the same pattern as the open functions, but with `updateEntity` and `deleteEntity` respectively.

Link service plug-ins

As we already mentioned, for an extension to be functional, it needs to provide two plug-ins: a data plug-in, and a gateway.

word_plug-in

The `word_plug-in` is the data plug-in, and responsible for defining the concepts of a `WordResource` and a `WordSelector`. The `WordResource` implementation is trivial, as it only sets the uri of the document, and defines its mime-type as `text/word`. `WordSelector` contains the raw identifier of a selector. As mentioned, the raw representation of a selector is the identifier of a document binding, and is contained in the `XPointer` variable. The class also provides the basic getters and setters, as well as an overridden `equal` operation, to make sure selectors are not duplicated in the database. A `toString` method is also provided.

gateway.word

This package contains the actual logic for handling the communication between Microsoft Word and the RSL core. The most important functions are `openDocument`, `updateView` and `deseraliseSelections`.

```
    @Override
    public JSONObject openDocument(Resource res, HashSet<Anchor> anchors, Anchor
↪  entityToHighLight)
```

Listing 4: Signature for openDocument

```
    @Override
    public  JSONObject updateView(Resource resource, HashSet <Anchor> anchors, String
↪  operation, boolean status)
```

Listing 5: Signature for updateView

```
    @Override
    public HashMap<Entity, String> deseraliseSelections(JSONObject addSelectionsCommand)
```

Listing 6: Signature for addSelectionsCommand

`openDocument` is either called when a link is clicked, which has a Word document as target, or when the Word add-in used the `isOpen` command. In the function, a new Javascript object is constructed, containing all the relevant information for the external application. It contains the resource, as well as the selectors and an optional highlight. The selectors themselves are Javascript objects, containing their sources and targets. As can be seen in listing 7, we can put additional information in the JSON object if we wish, such as the name of a link (currently not implemented in the backend). The optional highlight is only provided if a link has the Word document as target. The highlight is just another selector. All the required information (resource, selectors, anchors, sources and highlights) is provided by the RSL core.

```java
for(Entity entity: a.getSources()) {
        JSONObject source = new JSONObject();
        source.put("id", entity.getId());
        source.put("mime", entity.getMime());
        anchorSources.add(source);
}
```

Listing 7: Adding all sources information to our return object

`updateView` will be called if a the creation of a link was successful, or if a link pointing towards a Word selector was deleted, or if a selector to which a Word selector was pointing was deleted. The function itself follows the main logic as `openDocument`, and will serialise the resource and its (updated) selectors to a Javascript object.
`deserialiseSelections` is called when new selectors are created. The function will then transform the received Javascript objects into proper `WordSelector` objects, which will then be given to the RSL core to add.

```java
for(int i=0; i< length; i++ ) {
        JSONObject selection = (JSONObject) selections.get(i);
        String key = selection.get("status").toString();
        JSONObject wordSel = (JSONObject) selection.get("selector");
        WordSelector wordSelector = new WordSelector((String) wordSel.get("xpointer"));
        returnedSelections.put(wordSelector, key);
}
```

Listing 8: Deserialization to WordSelectors

5.3 Microsoft Powerpoint

Microsoft Powerpoint Add-in

As the entire user interface is identical to the Microsoft Word add-in, this doesn't need any further explanation. It should be noted that unfortunately, in Powerpoint we cannot use Office bindings, as they are limited to Microsoft Word and Excel. As a replacement, we opted to use the slidenumbers, since they are the finest grained control structure available to the API. This implementation detail is the only differentiating feature compared to Word add-in.

Link service plug-ins

powerpoint_plug-in

The `powerpoint_plug-in` implements the `PowerpointResource` and `PowerpointSelector` classes. Just as with Word, the `PowerpointResource` is trivial, and near identical to `WordResource`, with a different mime-type (`text/powerpoint`). The `PowerpointSelector` class also uses the `XPointer` variable, but now references a slide selection (e.g. 1-4), to indicate the selected slide range.

gateway.powerpoint

The implementation of the gateway is identical to the Word gateway implementation, as the differences in selector are only visible on the data plug-in layer, not to the gateway. This is one of the clear advantages of the chosen link service architecture.

5.4 Deployment

Office add-ins have multiple options for deployment, consisting of the official Office Store, a corporate add-in catalog or a shared folder. Deploying to the official Office

Store is a clear possibility, but brings with it several hurdles that need to be taken. After a new account has been created, it has to go through an approval process, which includes identity validation as well as account information quality checks. Once the account has been approved, the add-in can be submitted for approval. Once approval has been given, the add-in would be available in the Office Store. Unfortunately, this is a very time consuming process, which we did not undertake. A corporate add-in catalog can be created using SharePoint, which would be a trusted index for all Office applications on a corporate network. The same principles goes for a shared folder, but with a significantly lower overhead. For both, it boils down to a method to publish the manifest file, which contains all the required information for the plug-in to function. The actual add-in (the HTML and Javascript files) can be published on any existing webserver (such as Apache, NGINX or IIS).

6 Youtube Plug-in

To show even more clearly that the link service architecture allows for quick addition of new document formats, we chose to add a Youtube plug-in, which allows the user to select a time slot within a Youtube video as selector.

Google Chrome Extensions

To achieve this functionality, we chose to take advantage of the extensive Google Chrome extensibility. Chrome provides an extensive API to add content (javascript or css) to existing webpages, or to add tooltip-like windows (which are full-fledged HTML pages). A Google Chrome extension has two main parts: a manifest file, which declares the metadata about the plug-in, including the required permissions, as well as the intents.

```
{
  "manifest_version": 2,
  "author": "Tom Strickx",
  "name": "LinkService for Youtube",
  "description": "Create multidirectional links using the LinkService on Youtube",
  "version": "1.0",

  "browser_action": {
    "default_icon": "icon.png"
  },
  "content_scripts": [
    {
      "js": ["parse_uri.js", "document_start.js"],
      "css": ["plugin.css"],
      "matches": [ "*://www.youtube.com/*" ],
      "run_at": "document_start"
    }
  ],
  "permissions": [ "*://www.youtube.com/*", "storage", "declarativeContent" ]

}
```

Listing 9: Example manifest

Since our plug-in is intended for Youtube, it was a clear a content-script was to be used. By declaring our Javascript and CSS files in the `content_scripts` section of the manifest, the Javascript and CSS will be loaded at runtime into the declared pages: ``*://www.youtube.com/*''. The extension itself consists of two Javascript files, and 1 CSS file. `parse_uri.js` is a small MIT-licensed script which offers some more advanced URL parsing, which offered some better usability, but could be dropped. `document_start.js` on the other hand, contains all the logic and DOM-tree manipulation code to make the plug-in work. The script itself is called by adding an event listener to the current document for `DOMContentLoaded`, which means the attached function will execute once the entire webpage is properly loaded. After the page is loaded, we grab on to the youtube player, which is contained in a div called `player-api`. It should be noted, that while Youtube has an extensive API for embedded videos using the iFrame postMessage API, this is unfortunately not the case for videos on Youtube itself. By attaching a variable to the class `video-stream` within `player-API` we do have access to the HTML5 `<video>` object, which also offers several functions and attributes which we can use for our plug-in, but are not officially supported by Youtube, which unfortunately means usability might suffer if/when Google decides to revamp Youtube, and change DOM object ids or classes. In HTML5, an HTMLMediaElement, which is the abstract interface for both audio and video, has several properties, some of which can be modified, such as `currentTime`, which allows us to set the time (in seconds) of the video.

```
document.addEventListener('DOMContentLoaded', function() {
        var youtube = document.getElementById('player-api');
        var player = youtube.getElementsByClassName('video-stream').item(0);
        var linkDiv = document.createElement("div");
        linkDiv.setAttribute('id', 'link_container');
        var linkInfoDiv = document.createElement("div");
        linkInfoDiv.setAttribute('id', 'link_info_container');
        var buttonDiv = document.createElement("div");
        buttonDiv.setAttribute('id', 'button_container');
        youtube.appendChild(linkDiv);
        youtube.appendChild(linkInfoDiv);
        youtube.appendChild(buttonDiv);
```

Listing 10: Initial UI creation and video extraction after content is loaded

Once we are connected to the link service, again using websockets, the link service has total control over the current window, as long as we remain with the youtube domain. Just as with Office, we define an `onmessage` event listener, which provides our `parseMessage` function with the received message. `parseMessage` will then do an initial check if the message is a command, at which point it is passed on to `parseCommand`. If the message contains the `open` command, we will again do the parsing of the provided resource, selectors and optional highlight. This time, if a highlight is provided, it is in the format of a timespan, e.g. ”320-410”. This chains to `gotoTime`,

which will set the `currentTime` to 320 (seconds), as well as add an interval, which will monitor the `currentTime` property to pause the video once it hits 410.
The same principles for adding links from Office also apply to Youtube: since multidirectional links with many-to-many relations are a must, multiple start (and connected end) -points can be added to a linkqueue, which will then be submitted entirely to the link service. Unlike with Office, videos cannot be modified, so no tagging information regarding selectors can be embedded, so to list selectors, we are entirely reliant on the information provided by the link service. But the same principles also apply here: a selector can have multiple targets, which can be opened, updated or deleted. These UI elements are all displayed in-line using the Youtube CSS to be as non-obtrusive as possible, while still maintaining maximum usability, as can be seen in Fig. 4. Contrary to the limitations imposed by the Office API, this plug-in is capable of opening new (Youtube) resources in the same window, making for a seamless user experience.

Link service plug-ins

youtube_plugin

This data plug-in provides a `YoutubeResource` and a `YoutubeSelector` for the Youtube gateway. The `YoutubeSelector` in this case contains a `Time` variable, which contains a list with two values: start and end. This also means the constructor is slightly different from the previously implemented Selectors, which will also have minor consequences for our gateway.

gateway.youtube

This gateway will be attached to the `multimedia/youtube` mime-type. Most of the logic can be inherited from the already implemented other gateways, with some minor changes regarding the selectors. We no longer mention `Xpointer`, and the constructor now takes a (parsed) array, constructed from the `x-y`-format string. Other than that,

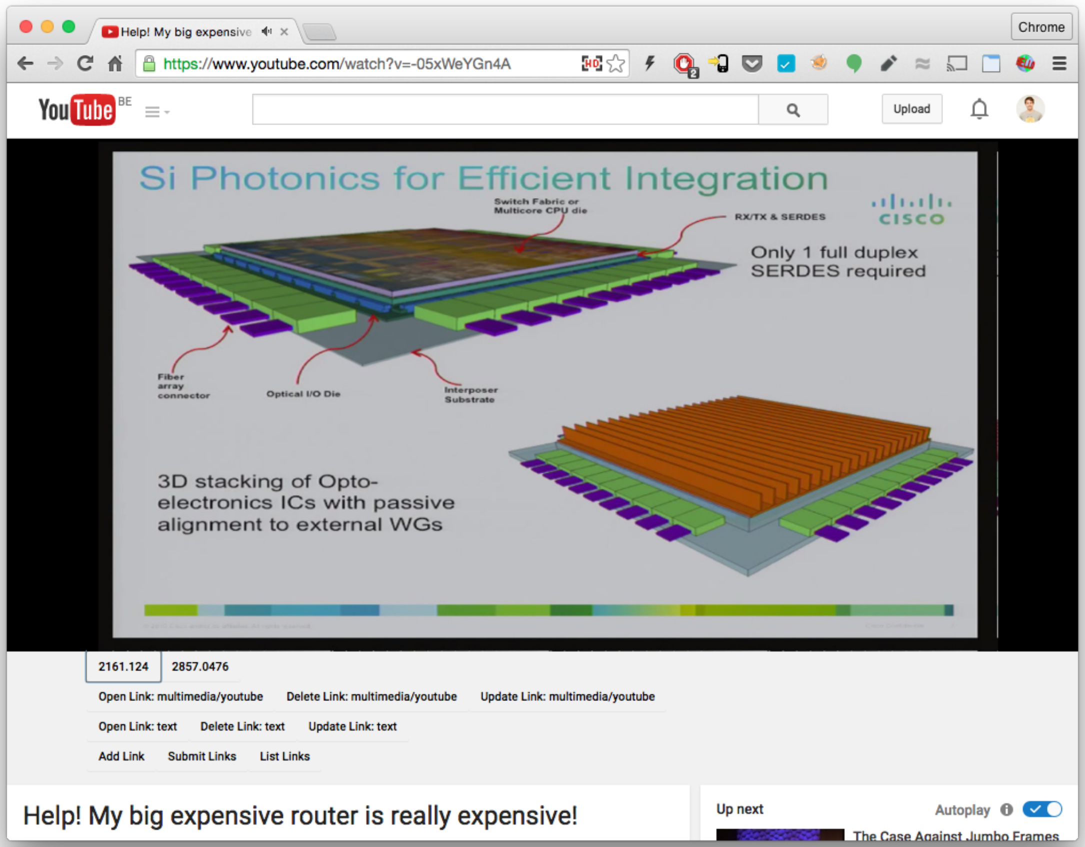
Help! My big expensive
Chrome
https://www.youtube.com/watch?v=-05xWeYGn4A
YouTube BE
Upload
Si Photonics for Efficient Integration
Switch Fabric or Multicore CPU die
RX/TX & SERDES
CISCO
Only 1 full duplex SERDES required
Fiber array connector
Optical I/O Die
Interposer Substrate
3D stacking of Opto-electronics ICs with passive alignment to external WGs
2161.124
2857.0476
Open Link: multimedia/youtube
Delete Link: multimedia/youtube
Update Link: multimedia/youtube
Open Link: text
Delete Link: text
Update Link: text
Add Link
Submit Links
List Links
Help! My big expensive router is really expensive!
Up next
Autoplay
The Case Against Jumbo Frames

Figure 4: Youtube User Interface

logic can remain identical.

6.1 Deployment

Previous versions of Google Chrome allowed users to download any extension, and install it in their browser. Unfortunately, due to security concerns, this is no longer possible. To install an extension, it has to be published to the official Google Chrome webstore, or the user has to enable developer settings to load an unpacked plug-in. A Google Chrome extension itself is nothing more than a zip-file (with a custom extension (.crx)), containing the manifest file, and the other required files, such as images, Javascript, HTML and CSS. To enhance security, the zip-file is also signed with a private key, which ensures that malicious extensions cannot pose as the genuine article. While the submission process is not as cumbersome as with Microsoft, it does require a one-time $5 developer signup fee. If a user has enabled developer settings, they can load an unpacked extension using the Chrome Extension menu.

7 Adobe Acrobat Plugin

Adobe provides two very extensive software development kits, one in `C++`, the other in Javascript. While both offer an immense potential to the developer, both SDKs are poorly documented, especially the Javascript SDK. Initial attempts were made to use the Javascript SDK to write a plugin, but unlike Google Chrome or Microsoft Office, the Javascript engine provided is severely limited in functionality, and only offers some basic Objects, which severely limits the possibilities. For example, communication to an external service is only possible using the (antiquated) Simple Object Access Protocol (SOAP), which is currently not an available protocol in the link service. The documentation, while extensive, provided multiple examples which were non-functional. While Adobe recommend using the Javascript API for plugins that wish to communicate with web services, we found it severely lacking to do so. We proceeded to experiment with the `C++` SDK, which was better documented, with running examples. Unfortunately, as with the Javascript SDK, we encountered numerous issues regarding communication with external services. Adobe provides support for Interapplication Communication (IAC) through Object Linking and Embedding (OLE) and Dynamic Data Exchange (DDE) on Windows. IAC offers methods and events which are wrappers for the core API calls in the Acrobat SDK, making it possible to control Acrobat in other applications. While this seems like a viable route for the future, the current iteration of the link service does not support DDE. To work around this issue, we tried to, at first, use several third-party libraries for websockets. While compilation finally succeeded after multiple issues regarding library linking and header inclusion, Acrobat did not run. Unlike with the development of the Office Add-ins, this was an incredibly difficult issue to debug, as we could not attach a debugger to Acrobat, as no debug symbols were provided. Adobe suggest writing logging information to a file, but this would constitute basic print-debugging, and would be very time consuming. As a final resort, we tried to implement communication using the Winsock2 library, which offers socket programming to Windows applications. This unfortunately did not result in a successful implementation, as again multiple issues were encountered in the cooperation between the socket SDK and Acrobat SDK. Further investigation into this issue was deemed too time consuming, and the plugin was scrapped. We did implement the data plugin and gateway for the linkservice, but as the data plug-in, and part of the gateway logic, are dependent on the representation chosen in the Acrobat plug-in, they will not be explained in this document.

8 Difficulties

Multiple issues were encountered while both developing the Office and Youtube plug-ins. In this section, we will try to properly document these, and suggest possible solutions or document the workarounds that we used to resolve them.

8.1 Communication

We opted to use websockets for all the implemented plug-ins, as it was the most obvious choice when working in a browser environment. This did have an unfortunate side-effect related to SSL. Microsoft forces Add-in developers to host the web content of the Add-in on SSL-secured webpages, which in itself is an excellent move, but this means all related communications have to happen using SSL. This includes the websocket communication. While we can (temporarily) accept an invalid (self-signed) certificate for a regular webpage (unless HTTP Strict Transport Security (HSTS)[1] is implemented, but

[1] `https://tools.ietf.org/html/rfc6797#section-5.2`

this is beyond the scope of this document), this is not the case for websockets. If we want to use SSL-enabled websockets, it has to offer a valid SSL certificate. At this time, the websocket implementation in the link service does not offer secure websockets, which made communication impossible. To work around this issue, we used a remote, publicly accessible server as an SSL terminator using NGINX. NGINX on the server would offer a valid SSL certificate for a domain name, and proxy the websocket communication channel to the link service. This is entirely transparent, and has no influence on the user. While this workaround works as a test case, it is not a viable deployment strategy. To resolve this issue permanently, secure websockets should be implemented in the link service, together with registering a domain name, and SSL certificate for said domain name. By pointing the domain name to 127.0.0.1 (localhost), users can run the link service on their local machines without the need for external services, while still being able to make use of secure websockets.
The same issue was encountered with Youtube, which also implements HSTS, making it impossible to downgrade the connection from https to http. As with Office, the external server was used as SSL terminator to provide fully functioning secure websockets to the plug-in.

8.2 Microsoft Office

The Javascript Office API is very extensive, but at the same time, somewhat limited. Due to security concerns, the available controls over the document are limited, especially finer grained tools such as word selection. While the `C#` and .NET API offer the possibility to get exact word locations from a selection with simple integer values, this is not the case in the Javascript API, and we had to resort to using bindings in Word, and coarser grained selections (slides) in Powerpoint to work around this. This was an unfortunate side-effect of the choice made to go for a broader supported API. The same security concerns, as well as multi-platform restrictions also made it impossible for an add-in to open another document, since this would have different implications in the Office 365 application for example. We worked around this by prompting the user to open the file manually, and to try again. We furthermore encountered either a bug, or a discrepancy in the documentation. To make current selectors visible within the document, we wanted to insert either HTML or Open XML in the document to highlight the text, to make it stand out to the user. While the API documentation clearly states it is possible (and indicates so by giving an example)[2], this was not the case when implementing this feature, as we encountered an incompatible coercion type error[3]. This meant we could only insert plaintext into our bindings, making markup impossible. Unfortunately, we did not find a workable solution or workaround to this issue, besides scrapping the entire project, and reimplementing it using the `C#` and .NET API, which would pose its own major issues.

8.3 Youtube

As was already mentioned in the Youtube plug-in section, Google does not provide an API for Youtube videos on youtube.com itself, only for embedded videos on your own webpage using iFrames. Fortunately, Google has recently (January 2015)[4] decided to use the HTML5 video player as default, officially discarding the ageing Flash player. This gave us the possibility to access the raw video interface, and use that to our advantage. This would not have worked as well if Flash was still the default, which does not have these easily accessible properties. The downside is that this is entirely unsupported, and can break with the smallest change in the DOM structure of the Youtube website. Previously Youtube also offered the possibility to specify an end time to a clip, indicating the selected clip within the entire video with a small yellow bar on top. This feature

[2]https://msdn.microsoft.com/en-us/library/office/fp161120(v=office.15).aspx)

[3]https://stackoverflow.com/questions/23294527/correlation-between-bindingtype-and-coerciontype-in-the-javascript-api-for-offic

[4]http://youtube-eng.blogspot.jp/2015/01/youtube-now-defaults-to-html5_27.html

has apparently been deprecated with the dismissing of Flash. This meant that we had to insert an interval, calling a time check function to check if we were at the end of a selected clip, while this is a minor nuisance, the lack of proper clip indicator is a clear defect in the user interface we could provide.

9 Conclusion

In this report we have shown our intent to implement four additional plug-ins for the Cross Document Link Service, offering advanced linking for Microsoft Word, Microsoft Powerpoint, Adobe Acrobat, and Youtube. We succeeded in implementing three plug-ins, as implementation of the fourth was severely hindered by both time constraints as well as poor documentation. We have proven the existing link service offers a viable platform for third-party plug-in implementation, extending the functionality of the platform considerably. By implementing these plug-ins, significant experience was obtained in multiple development environments, as well as learning experience with multiple SDKs in different programming languages. Due to the encountered issues, we now have a better understanding of the link service and its architecture, as well as a better understanding of communication restrictions and SDK issues.

9.1 Suggestions

The current link service communication protocols are very reliant on the JSON transport system, which is locked in in multiple levels of the application. We think it would be better if a better separation of concerns was made, where the data plug-ins and gateways are transport-agnostic. This improvement would make it much easier for external applications to use other transport methods, such as SOAP or YAML Ain't Markup Language (YAML). This is currently not possible, even if a data and gateway plug-in were written to support it, as JSON is currently tied to all communication protocols. As already mentioned in the difficulties section, the websocket server should provide secure

websockets, using a valid certificate.
Currently, the link service is user-local, meaning that sharing of documents which have these advanced links is futile, as all information is contained in the database of the application. Redesigning the link service as a proper web service would offer the ability for documents to be shared among users of the service, greatly increasing the usability.

References

DeRose, S., Maler, E., Orchard, D., & Trafford, B. (2000). Xml linking language (xlink). *Working Draft WD-xlink-20000221, World Wide Web Consortium (W3C)*.

Microsoft. (n.d.). *Privacy and security for office add-ins.* Retrieved from `hhttps://msdn.microsoft.com/en-us/library/office/fp161165.aspx?f=255&MSPPError=-2147217396`

Signer, B., & Norrie, M. C. (2007). As we may link: a general metamodel for hypermedia systems. In *Conceptual modeling-er 2007* (pp. 359–374). Springer.

Tayeh, A., & Signer, B. (2015). A dynamically extensible open cross-document link service. In *Proceedings of wise 2015, 16th international conference on web information system engineering.*