Testing in Production: Enhancing development and test agility in Sandbox environment

Xiudong Fei
xiufei@microsoft.com

Sira Rao
sirarao@microsoft.com

Abstract

Developing and testing an application hosted in a sandbox environment presents unique challenges. Development and testing agility is slowed down when validating such applications in production environment. Production environments are complex, have restricted permissions for modifying and using the environment, are costly in terms of deployment and upgrades, and make it difficult for applications to capture relevant logging information and troubleshoot in real-time. To test such applications in production, the problems are aggravated when trying to validate implementation changes, gather information such as logs or if there is a need to troubleshoot the application at runtime as it would require updating the binaries on the back-end or performing in-process debugging by the developer. These problems are also observed when the Test and Ops teams need to deploy test environments to validate the application. Repeatedly setting up test environments is costly and it is hard to simulate a true production environment. All of these constraints slow product development, testing, troubleshooting and thereby delay achieving quality levels needed for release.

This paper introduces a detour concept for an application that is hosted in a sandbox environment. We share the experience of validating the Lync Web application in the context of a Silverlight sandbox environment. In this paper we discuss the tool and framework created to address the above mentioned challenges. We also discuss how we used this tool to intercept Silverlight binaries, modify them and re-direct this to a browser session that hosts the application under test. Additionally, the tool can remotely manipulate objects in the Silverlight application through the use of scripts. With the objects available at hand, we can programmatically manipulate them for out-of-process mode debugging and also use this mechanism as alternate UI automation framework. The tool can be used to enable logging, inflate log size and for product error handling and security testing through fault injection. Using this tool in test and production environments and based on cost of deploying simple test environments, we conservatively estimate a savings of two person-days per month. The paper concludes with lessons learned by our team throughout the development and deployment of this tool, and includes information on practices other teams can implement to achieve similar results.

Biography

Xiudong Fei has been a test engineer at Microsoft Lync group for the last 4+ years. His passion is to create new ways of testing to have business impact, and have fun.

Sira Rao is a Test lead at Microsoft. He has worked on the Unified Communications products at Microsoft for over 7 years. He is passionate about building high quality products that excites customers.

1. INTRODUCTION

We have an opportunity to improve development and testing of applications that are hosted in a Sandbox environment and need to be validated in production environments. Production environments have a high degree of complexity, have restricted permissions for modifying or using the environment, are costly in terms of deployment and upgrades and make it difficult for applications to capture relevant logging information and troubleshoot in real-time. The development and testing agility slows down when validating such applications in a production environment. It is difficult to test such applications in production when trying to validate implementation changes, obtain logs, or to troubleshoot the application at runtime. In addition, product teams must fall back to testing an application in production to help find problems at scale. The scale issues that are usually found in a production environment cannot be duplicated within a test environment. All of these constraints slow product development, testing, and troubleshooting and thereby delay achieving quality levels needed for release.

This paper discusses the tool that we created to address these challenges. We consider the case of our application, Microsoft Lync Web App (LWA), hosted in a Silverlight sandbox environment, and discuss the problems with validating applications in production environments through the introduction of a tool we named SilverlightDetour, that is based on a detour concept. Our Lync Web application needed to be validated for quality in test and more complex production environments and our SilverlightDetour tool allows us to overcome the restrictions of the production environment so as to validate the application under test.

In addition to using the tool, we also actively encouraged “dogfooding” of the Lync Web application and used the tool to inspect and manage the running of our application. In this paper we also provide background on Testing in Production concepts and how they apply to validating applications within a sandbox environment, detail the issues encountered while validating the sandbox environment application and our application in production, and then present a specific project based on a detour concept that addresses the issues related to testing of sandbox environment applications in production environments. We conclude with the lessons we learned, how our solution addresses the issues relating to testing in production and how other teams can benefit directly from our work.

The following terms are used throughout the paper:

- *Dogfooding:* This concept is when product team and others in a company actively use and identify issues in a product prior to release. This practice has been in use in Microsoft since 1988.
- *Microsoft Lync:* The next generation of Microsoft’s unified communications software that enables people to connect in new ways, anytime, anywhere.
- *Microsoft Lync Web App:* This is a real time communications, collaboration client application that is a part of the Microsoft Lync family of products (including server components and clients).
- *Topology:* We refer to this term when we describe the network configuration that consists of various computers and connections between them. In the case of Microsoft Lync Server topology, it consists of the computers running different server roles such as Front end, Edge, Conferencing servers as well as other peripherals such as load balancers, gateways and connections such as switches, WAN links etc. In the paper we will also refer to environments as topologies.
- *Product Package:* The product or application binaries that are retrieved from a back-end system
- *SilverlightDetour:* The tool that is central to the paper, helping to solve issues of testing in production.
- *Remoting:* Usually referring to .NET Remoting, this allows client applications to use objects in other processes on same computer or any other computer reachable over the network or allows for objects in same process to interact with one another across application domains.

2. BACKGROUND

In this section we discuss what it means to test in production, what we consider as sandbox environments, the application and the tools we employ in the context of sandbox environments and testing in test and production topologies.

2.1. What and Why – Testing in Production

Testing in Production ("TiP" in short) uses production environments and topologies to validate a system or application utilizing functional, security, performance and other classes of test. For client teams that develop applications that depend on back-end systems, such as web servers, the application would be validated in test and possibly a pre-production environment before release to production. However, product teams would fall back to testing an application in production to help find problems at scale. The scale issues that are usually found in a production environment cannot be duplicated within a test environment.

2.2. Sandbox Environment

Sandbox environments typically provide a tightly-controlled set of resources for guest programs to run in, such as disk space and memory. These environments are usually closed and restrict the program or application, which is hosted in this environment, from having access to system resources and from what the application can modify. One example of a sandbox environment is the Silverlight application framework.

2.3. Silverlight

Silverlight is a web-browser plug-in and an application framework that enables interactive media experiences and rich business and immersive mobile applications. Silverlight can be hosted within a browser or as part of another stand-alone application. Client applications that are hosted in the Silverlight environment are required to be compiled with Silverlight libraries. Since our client application required browser support, we used a browser to host our application. This meant that the Silverlight applications had to be delivered from a back-end system.

2.4. Product or Client Application

The product or client application we refer to in this paper is Microsoft Lync Web App (LWA). This application is compiled and built for Silverlight and hosted within the Silverlight Sandbox environment. The application has UI and platform components and communicates with back-end server components. To validate LWA application requires that we deploy the server components that LWA depends on directly and indirectly.

2.5. Tools Available for TiP

We needed tools to help with testing in production to overcome some of the obstacles of production environments such as permissions to upgrade, complexity of deployment and difficultly with troubleshooting. There are several tools that enable intercepting network traffic and allow inspection and manipulation of the underlying traffic. These are useful in manipulating HTTP specific traffic, while others can also manipulate lower level protocol traffic. Some of these tools are in the public domain and are well known:

Fiddler – A web debugging proxy which logs all HTTP(s) traffic between a computer and the Internet and allows you to inspect HTTP(s) traffic, set breakpoints, and make changes to incoming or outgoing data.

Middleman – This is an advanced HTTP proxy server with features designed to increase privacy and remove unwanted content.

2.6. Test Topologies

In order to develop and test software and validate it, the application and its back-end must be deployed. Usually the validation consists of unit tests, functional tests, integration tests and end-to-end scenarios. To validate functional, integration, and end-to-end tests we need to deploy test environments that mimic production environments. These test environments or topologies are usually maintained by product teams or a smaller engineering team that services one or more products. In either case, the engineers are forced to balance between deployment and maintenance, and actual product validation or maintaining core engineering services such as test harness, reporting, tracking failures, etc. Additionally, the resources to simulate production environments are not usually available to smaller product or engineering teams, leading to discrepancies between test environments and production environments. This implies that many scenarios would need to be revalidated at a larger scale, on more complex environments, and with multiple simultaneous users.

3. CHALLENGES OF TESTING IN PRODUCTION

3.1. Test environment limitations

Test topologies are usually very simple and use minimal resources. The goal is to validate core functionality and iterate on testing the application and back-end components. The role of the central service engineers is to ensure they provide the product team with simple topologies to validate some functional, integration tests and possibly end-to-end tests and thereby ensure that the product teams have a higher degree of confidence before rolling out the application and back-end changes to the next stage (possibly one or more of the following: “dogfooding” environment, pre-production environment and finally to production). The test topologies usually lack complexity and usage – scale cannot be represented and only simulated to a certain extent, lack integration of advanced components such as load balancers, edge servers. The scale of the test topologies could be enhanced to achieve better approximation of production topologies but the cost would be high for product teams. Additionally, if product team engineers maintain these topologies, they look to minimize their setup and maintenance time even if these machines are virtualized or some install steps are automated.

3.2. Complexity of Production Environments

Maintaining a copy of the production environment has its challenges. There are several quality gates an application and/or the back-end system has to pass through before a production environment can be updated. Installation and un-install require systematic and detailed steps in order to minimize affecting other systems and disruption to users. Additionally, the access and permissions to manage and enable logging of components is restricted. This is deliberate and is needed to ensure that productions systems are not meddled with in ways such as uploading temporary or not fully tested binaries, changing configuration or slowing the system or performance due to excessive logging.

3.2.1. On-Premise (on-prem in short)

Figure 1 shows a sample Production environment that supports high availability and a single data center. Some of the components of this environment include Active Directory, an array of front ends, load balancers, edge servers etc. Other components provide additional functionality in a Lync Server deployment. The Lync Web application depends on many components of the back-end system that are built by other product teams such as conferencing servers, front end and edge servers, PSTN gateways, and load balancers. These are deployed and setup by

other product teams who have expertise on the installation and configuration of these components.

The following topology depicts a Reference Topology for Production Environment: (Source: http://technet.microsoft.com/en-us/library/gg425939.aspx)

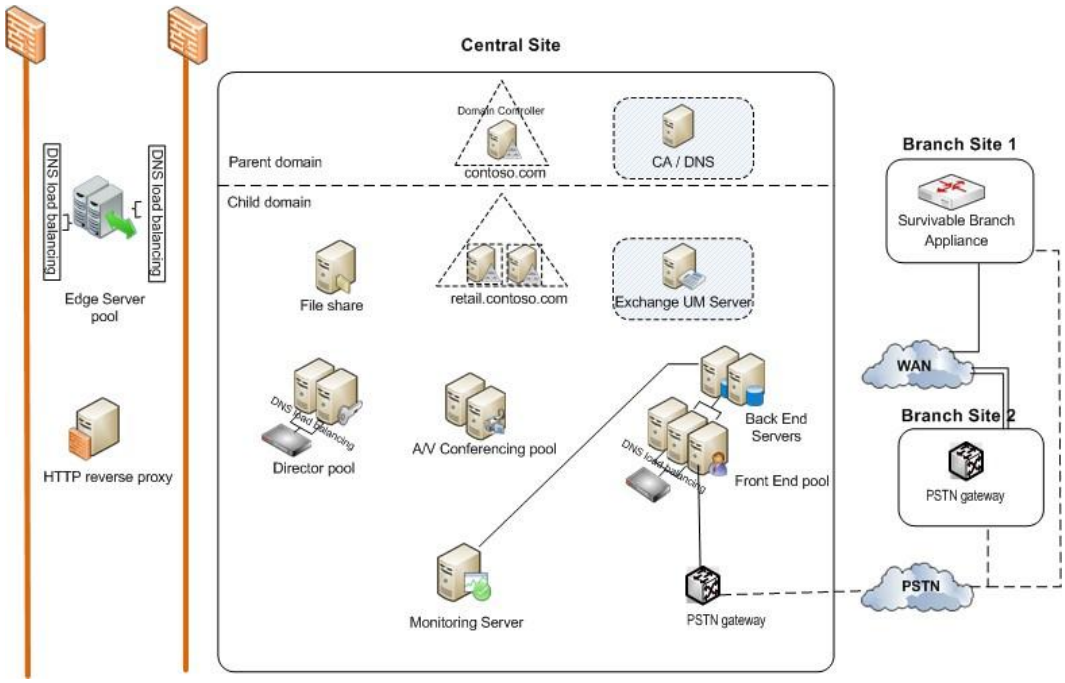

Figure 1 – Sample Topology diagram of Microsoft Lync Server showing the back-end components

3.2.2. Hosted

This type of environment or topology can be deployed by a partner or by the Office-365 service. The hosted or service topologies have more redundancy and scale and typically have more complexity than on-premise deployments, enable more users to use the back-end systems and stricter administrator controls.

3.3. Challenges for Testing in Production

As we discussed earlier, testing in production helps find problems at larger scale and the scale issues that are usually found in production cannot be duplicated within a test environment. However as we noted in the beginning of this paper, production environments come with high degree of complexity, restrict the access to only certain individuals to modify the back-end systems. This makes these topologies costly in terms of deployment and upgrades and makes it difficult for programs or applications (such as client applications) from capturing relevant logging information and troubleshoot in real-time.

3.3.1. Obtaining Logs and Log Size

Usually, the type and level of logging are fixed by default in the product (especially for sandbox environment applications) by the time it is released to production. To investigate and troubleshoot issues, the developer may need more details, so control over logging levels and components are needed. We also identified cases where it was not possible to get the logs if the product hung or exited. In this case, saving the logs to external storage was preferable.

Due to memory size limitations, however, the size of the logs that our application can write is limited. During our testing we observed many issues were missed as the logs saved by application in memory were overwritten due to the fixed size of the logs. In this case transferring the logs to external storage is preferable.

3.3.2. In-process Debugging

The usual debugging approach involves in-process debugging tools such as Visual Studio, which is attached to the specific target process, and the developer sets breakpoints where he or she can inspect and modify the states of variables. This type of debugging is valuable, however it requires stopping the process to inspect variables, see the call stack, etc. Sometimes the developer cannot interrupt the running process (especially when the debugging is timing-dependent) and in this case the only way to inspect the states of variables is through secondary means such as printing variable information into the log file or showing a notification. It is clearly evident that an out-of-process debugging mechanism would be helpful in this scenario.

3.3.3. Silverlight Cross-Domain Policy

Silverlight allows only site-of-origin communication for non-image and non-media requests. In order to enhance security of the backend systems, the server only allows download of the client application from the back-end server. While secure, this restrictive nature of the cross-domain policy reduces developer and test agility as it prevents validating private binaries and fixes. Although one option to overcome this is to run extra scripts to update the cross-domain policy on the back-end, this would not be feasible in production environments.

3.3.4. Validate prior to Deployment

With a production system, it is necessary to pass the quality gates prior to deploying an application or updating any of the back-end components. However, it is helpful sometimes to quickly validate the application in production environments without updating the back-end or even the application that is hosted on the back-end environment. This can provide increased confidence to the product teams having met required validations on production environment prior to updating the application in production.

3.4. Impact to Agility

As mentioned, the challenges of testing in production also directly affect the agility of product development and testing. When “dogfooding”, there is a need to iterate quickly in diagnosing the issue, making the code fixes, and validating the unit tests, functional and integration scenarios without updating the back-end environment or even the client application hosted on the back-end. We need to find and fix issues quickly so that we can encourage and ensure “dogfood” users that their feedback is valued. In certain scenarios, the UI of the application may need to be changed while the underlying platform and back-end server components remain the same. In this case we would like to validate the new UI interactions on the production systems. These changes are not directly possible in production systems due to the restrictive permissions and quality checks that one must go through to deploy new binaries so it does not impact the rest of the system and other users. This implies that product development, testing, “dogfooding”, and troubleshooting slows down the process, requiring more time to achieve desired quality levels.

When testing in production, the install and upgrade process require planning and maintenance, which delays system validation and testing, since deployment times, machine and upgrade issues can frequently delay validation. Ideally the changes made to applications will receive quick feedback from users and the process of fixing and validating can iterate quickly to achieve desired quality levels. Additionally, maintaining these production systems is costly in terms of time and resources.

4. SILVERLIGHTDETOUR SOLUTION

To solve the challenges of TiP, we used a two part strategy. The first was to encourage broad use of the application in production environment by employees outside the product team – which is termed “dogfooding”. This helped increase the development and testing agility, since the product teams iterated quickly on finding, fixing and validating issues that impacted the users in the dogfood group. The second part of our strategy was to create a tool we called SilverlightDetour, that helped us inspect and manage the running of the application in production or test environment, but remained inconspicuous to the users.

4.1. Dogfooding

When team members and other employees use their own product there is frequently a desire to enable better turnaround time between identifying the issue, and providing a fix and re-deploying the updated application. We solved the continuous Dev/Test iteration by encouraging the Dev to use the SilverlightDetour tool to validate their unit and functional tests against both test and production environments. In addition, the test team and central engineering teams pushed back on frequent topology updates and reduced this to monthly updates or when faced with breaking changes. This helped increase dogfooding of the Lync Web application on both test and production environments, and increased the use of the SilverlightDetour tool to inspect and manage the application.

4.2. SilverlightDetour

We needed to validate our Lync Web client application in test environments (simple) and in production environments, which are more complex. The SilverlightDetour tool allowed us to overcome the restrictions of the production environment by redirecting the communication through a proxy.

4.2.1. Overall Architecture

Figure 2 highlights how the SilverlightDetour tool and framework behaves. In the diagram, it shows how the product or application package from the server is intercepted and modified by SilverlightDetour, and the modified package is passed to the user where it is loaded in a browser session. The tool is implemented in C# and uses the .NET framework and runs as a standalone executable (.exe). The tool can be run on a machine that also runs the Lync Web client application, or the tool can be on a remote machine and configured to accept requests from another machine that runs the client application. We established a TCP connection between the tool running on a desktop machine and the client application within the Silverlight sandbox environment. This helped address the issue of collecting logs for and troubleshooting of the application, especially when testing the application in production.

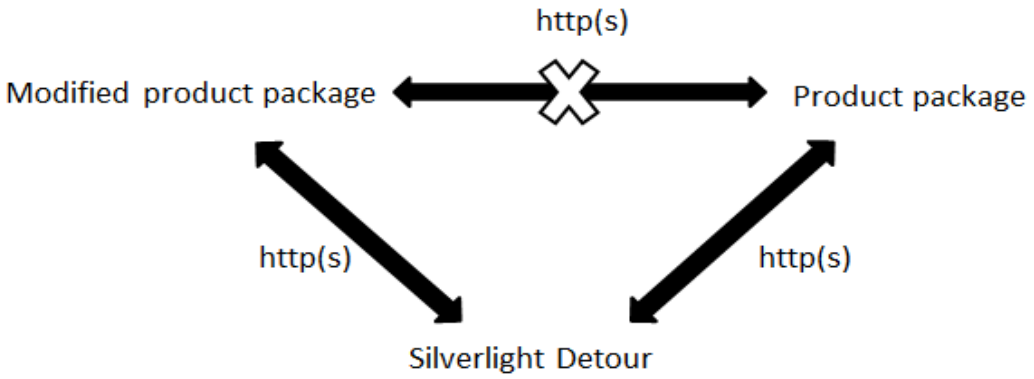

Figure 2 – Overall Architecture

4.3. Implementation Details of the Tool

4.3.1. JavaScript

Using the SilverlightDetour tool we were able to remotely manipulate Silverlight objects through the use of a TCP connection pipe between the Silverlight application and the tool (that is running on a desktop machine). This was done via a small piece of JavaScript that is injected into the HTML page that hosts the Silverlight application. When the client browser loads the Silverlight application, that JavaScript code is executed and creates a Silverlight object from a small piece of XAML. Within this Silverlight object a TCP connection is initiated with SilverlightDetour tool. The code snippet for the JavaScript is mentioned later in Sect. 4.4.7.

4.3.2. JSON

To manipulate objects and identify their type, we needed to serialize and de-serialize them. This was accomplished using a JSON serializer to extract objects to a string format, which is then sent and received using TCP transport. Before an object is sent across the TCP transport, it is serialized and packed into JSON format that can be transmitted across the network. At the other end of the TCP connection, the serialized data is read and is then de-serialized back to the actual object or type that it represents.

4.3.3. TCP connection

As mentioned earlier, a TCP connection is made between the Silverlight sandbox and the desktop machine. The Silverlight object makes a TCP connection to one of Silverlight's allowed ports (4502 - 4532). On this same connection, there are two kinds of data that are transferred: (1) to send and receive the serialized JSON stream and (2) to send logging message back to SilverlightDetour tool. The transfer of JSON stream was used in the manipulation of Silverlight objects and the log messages were used for the purpose of getting logs to the device and inflating the log size.

4.3.4. Use of Fiddler

We incorporated Fiddler's FiddlerCore library into the SilverlightDetour tool. This allowed us to perform HTTP/HTTPS traffic inspection, monitoring and modification with our tool.

4.3.5. Remoting for Silverlight

The SilverlightDetour tool passes objects between Silverlight sandbox and desktop and this is done in a similar manner to .NET Remoting mechanism –behaving as a sort of a .NET Remoting infrastructure for the Silverlight application. For this purpose we included some features of another internal tool that supports limited remoting for Silverlight. We used this remoting to manipulate objects as well as transfer logs from Silverlight environment to a hard disk on a local or remote machine.

4.4. How SilverlightDetour addresses the Problems of TiP

4.4.1. Improves Dogfooding

Since dogfooding was a requirement for most of the Lync family of products, we had to enable non-product team members to use and validate our Lync Web application in production. However, we also wanted to find a way to manage and inspect the application if any issue did come up during testing. One important feature of the tool is the ability to replace product binaries, a feature that was often used by those dogfooding our application. This helped us work with Dev and other Test members to increase adoption of SilverlightDetour tool by highlighting and encouraging use of this feature.

As mentioned in Figure 2, the application package from the server (back-end system) is downloaded via HTTP(s) and is then unzipped. The SilverlightDetour tool modifies the package manifest file to replace any existing assemblies or add any additional ones, re-zips the package and then sends the HTTP(s) stream back to the web browser session. The

browser loads the modified package and now the “most recent” versions of the client application can be validated against the same back-end system such as a production environment. This allowed us to validate code fixes and changes before broad roll-out on production environments.

4.4.2. Avoids Install/Uninstall pains

Since the application binaries can be replaced using the SilverlightDetour tool, the constraints of deployment and upgrading back-end systems and the delays involved in that can be by-passed to a large extent. In addition, the test team and central engineering teams pushed back on frequent topology updates and reduced this to monthly updates or when faced with breaking changes that affected our Lync Web application, such as between platform and server back-end components.

4.4.3. Overcomes Production Environment restrictions

Using the tool we are able to validate the application, obtain logs, troubleshoot application at run-time in production environments – both on-premise and hosted. This helped us avoid the restrictions and permissions relating to getting logs for application, troubleshooting application, replacing application binaries to validate code fixes and waiting on deployment upgrade cycles.

4.4.4. Inflating Log Size and Retrieving Logs

We establish a TCP connection between the tool and client application. Once the connection is established, users can dynamically disable or enable relevant log components, change the level of logging and extend the size of logs in real-time by making the entire disk available for log storage, rather than what’s available in memory and limited by the application. Using the SilverlightDetour tool we pass objects between the Silverlight sandbox and the machine running the tool as depicted in Figure 3. Retrieving logs is similar to inflating log size where we pass objects and transfer logs from sandbox to local machine.

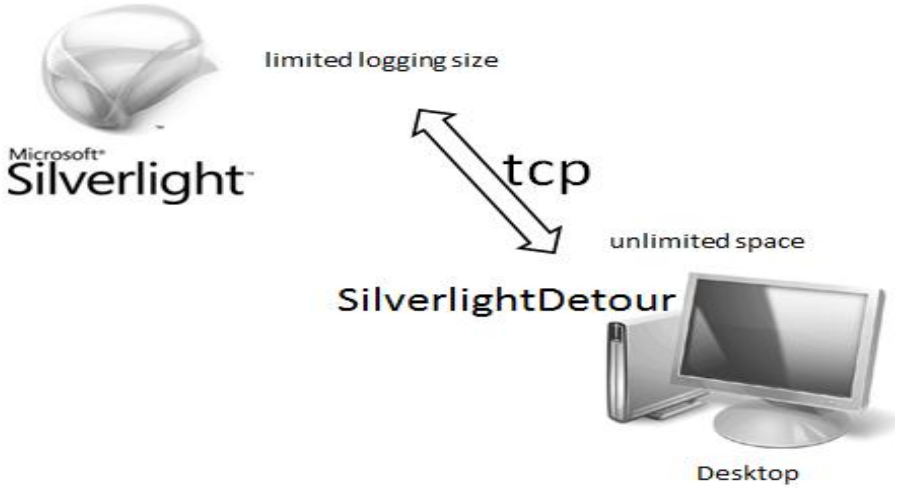

Figure 3 – High level design of Unlimited Silverlight Application logging

4.4.5. Debugging made easy

With the power of being able to pass objects between the Silverlight sandbox and desktop, it enabled useful and interesting scenarios for the product team. The communication and object passing mechanism allowed the application to pass objects by reference remotely between the Silverlight environment and the tool and we were able to inspect and modify the values of variables, the state of Silverlight objects, and to validate the behavior while the application is running. This allowed us to enable debugging on these objects by remotely controlling the

Silverlight UI elements and enabled out-of-process debugging during runtime without attaching to a debugger. In addition, we could also set breakpoints based on HTTP(s) request / response, which is hard in traditional debuggers.

In this example the SilverlightDetour tool manipulates the remote Silverlight objects in our client Silverlight Lync Web application. The figure and code snippet shows an example of out-of-process debugging using scripts such as PowerShell. The tool obtains a Silverlight object having type as text box (referred to as *displayName*), modifies the text property, inspects the value of '*TextAlignment*' property and modifies to right alignment. We can also use this tool to debug based on HTTP(s) request / response sent between client and back-end.

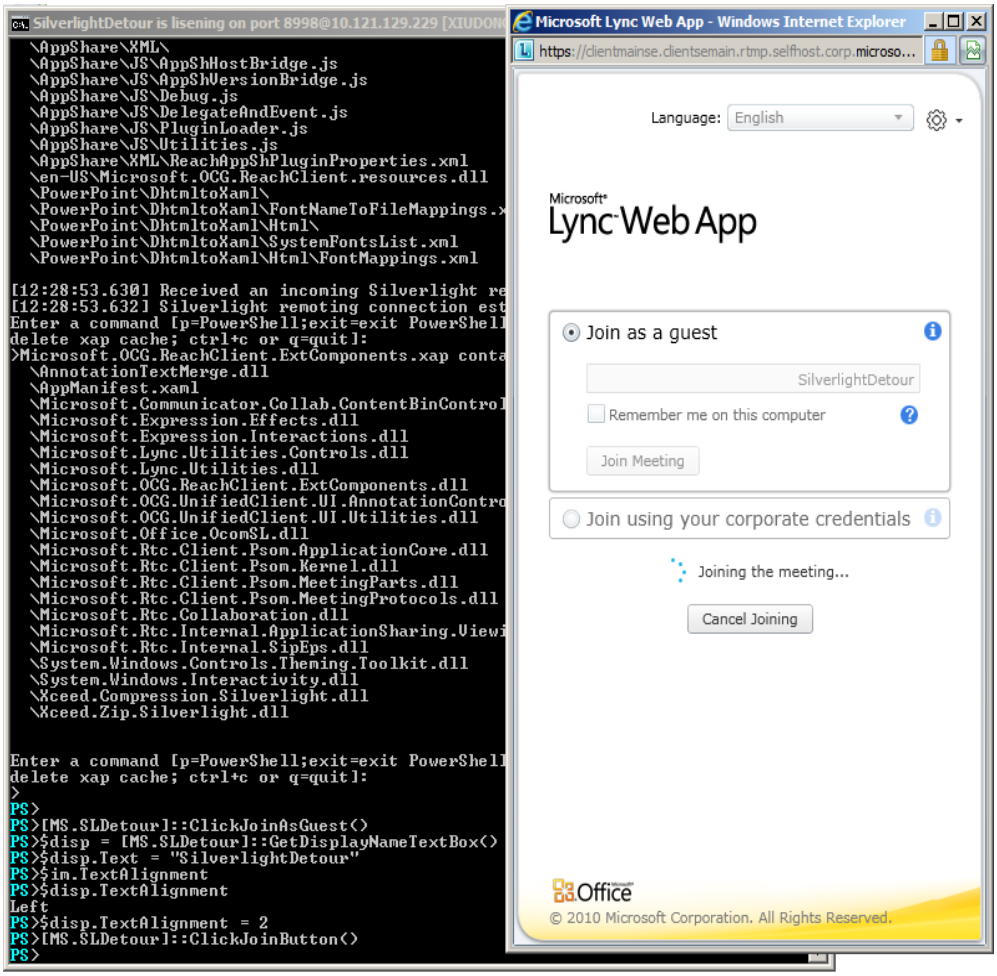

Figure 4 – Using Scripts (PowerShell) to control remote Silverlight objects

PS>$disp = [MS.SLDetour]::GetDisplayNameTextBox()
PS>$disp.Text = "SilverlightDetour"
PS>$disp.TextAlignment
Left
PS>disp.TextAlightment = 2

4.4.6. Facilitates Remote Assistance applications

Our tool also supports a lightweight form of remote assistance. If a customer needs help on using a feature of the product, the support team can start a TCP listener and accept the

incoming request from the customer. The support team can issue commands to highlight and modify the Silverlight UI elements and hence guide the customer in a simple and step-by-step manner. This also allows the support team to switch to debugging and inspecting the public and internal variables of the application should anything go wrong. This seamless move from Helping to Debugging is possible within a single tool, while it would be cumbersome to set up and use a traditional debugging tool just for the latter case.

4.4.7. Allows Programmatic Manipulation via Scripts

Silverlight does not normally allow remote manipulation of objects. To overcome this limitation, we had the idea of injecting JavaScript during the interaction between the client and back-end. The tool checks if a HTML page contains the Silverlight application by searching for 'object' node which contains type="application/x-silverlight-2". The tool then injects the following: "param node name = onload , value = CreateBridgeServiceLoader"

```
1.  <object "type="application/x-silverlight-2">
2.      <param name="source" value="product.xap" />
3.      . . .
4.      <param name="onload" value="CreateBridgeServiceLoader"/>
5.  </object>
```

Here the code snippet shows how to use JavaScript to create a Silverlight object and that object initiates a TCP connection to a certain machine (e.g. machinename.domain.com) and port (e.g. "4510")

```
1.  <script type="text/javascript">
2.      function CreateBridgeServiceLoader(sender, args) {
3.          var url = "tcp://machinename.domain.com:4510";
4.          var silverlightPlugin = null;
5.          silverlightPlugin = sender.getHost();
6.
7.          if (silverlightPlugin != null) {
8.              try {
9.                  var obj = silverlightPlugin.content.CreateFromXaml("<host:BridgeServiceLoader
    xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation'
    xmlns:x='http://schemas.microsoft.com/winfx/2006/xaml' xmlns:host='clr-
    namespace:Microsoft.RemoteBridge.Host;assembly=RemoteBridge.Host'
    ProxyServerAddress='" + url + "' />");

10.                 if (obj == null) {
11.                     alert("Failed to call CreateFromXaml to establish a Silverlight bridge
    connection.");
12.                 }
13.             }
14.             catch (exception) {
15.                 alert("Failed to create BridgeServiceLoader.");
16.             }
17.         }
18.         else {
19.             alert("Page does not contain Silverlight application.");
20.         }
21.     }
22. </script>
```

Figure 5 shows a sequence diagram detailing the interactions between the SilverlightDetour tool and the Web Browser and the Server (Back-end system).

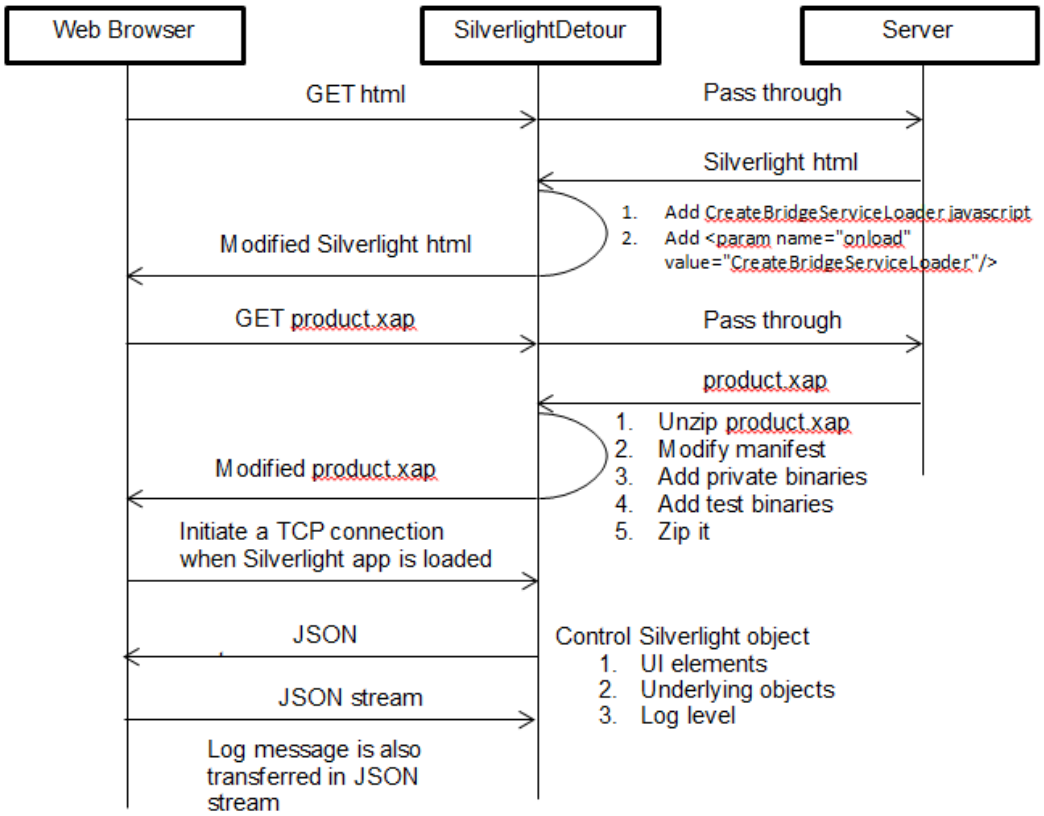

Figure 5 - Sequence Diagram showing actions between web browser, SilverlightDetour tool and server (back-end system)

4.4.8. Provides Alternate UI Automation Mechanism

Using the TCP connection between the Silverlight sandbox and Desktop and the ability to manipulate Silverlight objects, it is easy to manipulate remote Silverlight UI elements using Silverlight Class library. Figure 4 in sect. 4.4.5 shows an example of UI element modification using the tool.

Code snippet:

```
FrameworkElement rootElement =
Application.Current.RootVisual.Cast<FrameworkElement>();

static public TextBox GetTextBox(string name) {
        return rootElement.FindChildren().ByName(name).Cast<TextBox>();
        }

static public RadioButton GetRadioButton(string name) {
        return rootElement.FindChildren().ByName(name).Cast<RadioButton>();
        }

static public void ClickRadioButton(string buttonName)  {
        RadioButton button = GetRadioButton(buttonName);
        ToggleButtonAutomationPeer peer = new ToggleButtonAutomationPeer(button);
        IToggleProvider provider =
peer.GetPattern(PatternInterface.Toggle).Cast<IToggleProvider>();
```

```
        if (provider != null)
            provider.Toggle();
         }

    static public void ClickJoinButton()  {
        ClickButton("JoinButton");
        }
```

4.4.9. Bypasses Cross-Domain Policy

As mentioned in Sect. 3.3.3, to enhance security of the back-end, the server allows only downloading the client application from the server. This means that validation of private binaries against production systems would not be possible unless the policy is bypassed. SilverlightDetour is used as a cross domain policy server so that the tool returns a lower restriction policy when the Silverlight client requests for the cross-domain policy ("GET clientaccesspolicy.xml"). Hence the tool overcomes the limitations of the Silverlight sandbox environment while continuing to provide agility to the Dev/Test cycle. Below is an example of the policy the tool provides:

```
<?xml version="1.0" encoding="utf-8" ?>
<access-policy>
  <cross-domain-access>
    <policy>
      <allow-from http-request-headers="*">
        <domain uri="*" />
        <domain uri="https://*" />
        <domain uri="http://*" />
      </allow-from>
      <grant-to>
        <resource path="/" include-subpaths="true" />
      </grant-to>
    </policy>
  </cross-domain-access>
</access-policy>
```

4.4.10. Enables Fault Injection and Security testing

Given that we are able to manipulate the Silverlight objects as well as perform out-of-process debugging by tracking the HTTP/HTTPS requests or responses, we were able to manipulate error codes, change protocol schema elements, simulate hard to obtain errors and perform fuzz testing of the XML parser in interesting ways. However, for lack of time we covered only the mainline scenarios. An additional complication to using the SilverlightDetour tool was that our Lync Web application and back-end needed to support compression so that the LWA would work well on low bandwidth networks. We were able to test with compression as well as using the SilverlightDetour tool to negotiate not having compression. This was done by changing a success response to a compression request to an error thereby disabling compression for the duration of the browser session.

5. RESULTS

5.1. Dogfooding

We observed that the SilverlightDetour tool helped the product Dev team to resolve issues before code check-in by running unit and functional tests against test and production environments. The test team also used the tool to run functional and integration tests to identify issues. This

reduced the Dev/Test loop in terms of code check-in, testing, resolving issues, and verifying fixes. In addition, when the product team encouraged dogfooding by other individuals, issues were quickly debugged through the use of the tool and once again brought to the product team's attention.

5.2. Appreciation and Collaboration

The tool and framework that was developed by the product team members received a lot of positive feedback across diverse teams (Redmond, India development/test teams). In addition, we also received Peer Recognition awards for the benefit to the product and our partners in the development team.

As the SilverlightDetour tool gained more usage within Dev /Test teams, there were more suggestions and requests to add features and provide extensibility hooks. This required managing the requests in a similar manner as the requests for adding additional features to a product. This was resolved by having additional members contribute to the enhancement of the tool and by balancing priorities of tool development and product validation among the other product team members.

5.3. Improvements in Testing and Validation

The SilverLightDetour tool allowed for improved testing of the Lync Web Application in terms of functionality, stability, overall correctness of the application and of the back-end environment.

1) Ability to obtain logs when the application crashes, exits abruptly, or hangs due to application, network, or back-end issues
2) The tool allows teams, that use Silverlight, to validate the application package that is downloaded from back-end system
3) Ability to simulate errors and verify user experience related to scenario and error message
4) Improved testing of resiliency scenarios, which were not previously possible since we could not bring down the production systems to test error and fail-over scenarios

5.4. Cost Reductions and Productivity Gains

5.4.1. Dev/Test spend minimal time on Setup

By testing in production, we found the Dev team could stop spending time investing on topologies to test their client side functionality. Testing in production also benefited the product team test engineers as well as the central service engineering team since they spent less time setting up and updating the topology, dealing with topology setup failures and also communicating with admins / engineers on production environments.

Prior to using this tool, the development cost was one person-day per month, while the test team required two person-days per month. After using the tool these costs went down, and there was an increased use of the tool to test changes and bug fixes to the application against a Production environment before upgrading the back-end. If we were to consider deploying a topology that resembles production topology (as mentioned in sect. 3.2.1), the costs would go further up and would require a dedicated resource for managing the machines, deployment of the many server roles.

5.4.2. Affects Productivity in other tasks

With the cost savings obtained from spending less time on setup, installation and maintenance of topologies, the product team engineers (including the authors) were able to spend the additional time validating the quality of the product, automating more tests, and coordinating better testing through a wider set of users in the dogfood group. The overall productivity of the Dev team went up as they spent little or no time worrying about topology and the productivity of the Test team went up as they spent less time on routine and difficult tasks such as setup, deployment and maintenance of the topology.

5.4.3. Gains in Production systems

This cost cannot be estimated by us since it involves setting up things in an automated manner as well as running manual configuration changes and requiring one or more dedicated resources. We do estimate that by providing the client application to other individuals, we identified more issues at scale. For example, in large scale online meeting scenarios we observed sluggishness and high memory usage by the client application. We had fixes for these within a week and used the tool to verify the behavior in another large scale online meeting on production environment before rolling out the updates to production environment.

5.4.3.1. *On-Premise*

We primarily validated our Lync Web client application using the SilverlightDetour tool against this type of environment since many customers have the Lync Server deployed on their premises.

5.4.3.2. *Hosted*

We validated our application using the tool against hosted environments towards the latter part of the release. The same gains and costs are applicable here as these have more complexity than on-premise production environments.

5.4.4. Agility

Agility increased as the Dev / Test accelerated the rate of validating unit and functional tests, and troubleshooting and debugging issues. The product teams were quickly able to validate the application in production environments without updating the back-end or the application that is hosted on the back-end environment. This increased the quality of the application and the product was able to go out to the next gate (either for dogfooding or Production) having met a higher quality bar.

5.4.5. Benefits due to Remote Assistance and Debugging

The remote assistance and debugging features of the SilverLightDetour tool were demonstrated and used by the team members in a number of ways. One such instance was when we wanted to debug on a Mac machine and show how this can be used as a help mechanism. Having the Mac machine point to the computer running the SilverlightDetour Tool, we were able to perform basic troubleshooting on that computer. In fact, this tool can be used along with any of the major browsers that are supported on Windows OS and Mac OS. This type of run time debugging can help improve productivity of the product team (Dev or Test).

5.5. Lessons Learned

5.5.1. Getting people to use the tool / framework

With any new tool or framework the hurdle to cross is adoption of the new way to work. In this case, the product test team began using this tool and pushing back on topology upgrades. Simultaneously, we worked with the product Dev team to use the tool to validate changes before code check-in. After highlighting the benefits and seeing the tool in action, the product Dev team adopted the tool and gradually began evangelizing it among other Dev and test members involved in developing UI and back-end system for the products.

Eventually the tool became vital to development for validating their unit and functional tests, debugging remote issues, validating in production environments with minimal effort and improving the development to test to release cycle.

With the Dev team evangelizing the tool, the entire product team was more careful with checking in breaking changes that needed fixing or updating the tool and framework. This changed the perception of how Dev / Test teams worked so that we were working to mutually benefit the other and in turn help ourselves. The balance shifted more towards finding ways to utilize the tool to validate product quality and less towards routine server and topology maintenance and deployment.

5.5.2. Paid more attention to the network traffic

This tool and framework helped us understand the details of how our product worked at the protocol and network level. Product Dev / Test members became more aware of the working as well as understood underlying issues. This helped us improve quality of the product through fault injection means and manipulation of the UI – as detailed in sect. 4.4.10

5.6. Challenges Faced

Although the framework supports testing in production for the majority of the scenarios, there were some challenges that the product team faced with this framework. In these cases, the product team had to spend some time to upgrade/install a new version of the test topology, validate the quality and then work with admins and engineers to upgrade the dogfood, end-to-end, or production environments.

The challenges were mainly the protocol schema mismatch and timer changes to match sending / receiving data between the client application and the back-end environment. These changes forced the product team engineers to update the test topology as well as ensure that these changes are timed to occur before the next roll-out of the product (application and back-end) to the next stage viz. dogfooding and/or Production. While it would have been possible to "exclude" the breaking changes and compile the application to work against the older versions of the back-end (that reside in production environments), we found that this caused more confusion and had the negative effect of not validating the product correctly.

5.7. Implication to Test In Production

As customers move to having their back-end infrastructure hosted online (in the cloud), the need to validate at scale and in complex environments is a necessity. Deploying such environments is not possible by product team engineers or the smaller service engineering teams. Hence the rapid iteration of the client applications (in sandbox) will need to be validated against Production environments without updating the back-end systems or even the hosting environment for the application. This will require finding solutions to validate client application changes at scale and getting enough "dogfooding" of the product before actual production roll-out. Framework and Tools such as our tool will help accelerate the product development, testing, "dogfooding" and iterate to achieve the quality levels needed before release.

5.8. How Other Teams can Use this

We looked at other teams that employ client applications delivered through the server and observed the class of Web and plugin based applications. Although the tool and framework can be extended to pure web based applications, we primarily focused on applications that are dependent on a plugin such as Silverlight. Other teams / products that utilize such plugins can employ the tool / framework in their specific environments to validate their applications in production. In addition, these products can utilize the different aspects of the framework for purposes of troubleshooting issues, developing programmatic manipulation of the UI such as for UI automation or to test security by fault injection.

In demonstrations within the company, we have seen interest in exploring the benefits and extensibility of the framework. We state that there are no product team decisions on whether they plan to utilize this framework. However, we see benefit in utilizing our framework especially in client server environments such as Bing Maps and other Silverlight based products. In such applications we need to keep the back-end Production environment up and try multiple instances of the client application to validate functionality, user experience while ensuring access to vast databases, supporting multiple users, minimizing disruption to large systems and reducing communication around upgrades / maintenance.

6. CONCLUSIONS

We had initially set out to make our client application testable in order to improve the quality of the product, but found that there were bigger problems to solve. As our focus changed we observed that we could implement a framework that accelerated the product development and testing of applications in sandbox environment for Production environments. In addition we observed that we could iterate rapidly in terms of identifying issues, troubleshooting in real-time and validating fixes all in production environments. This allowed testing client application at scale and in complex environments while minimizing the need for product team engineers to be involved in setup or deployment of even simple topologies. Having client applications running in production or near production environments provided increased feedback to product teams through "dogfood" programs and in turn provided the users and customers with higher quality products via faster development and testing cycles.

In addition our tool had the interesting side effect in terms of providing additional range of testing techniques and use – such as using it for fault injection in security testing, programmatic manipulation using scripts that can be used as alternate mechanism for UI automation and providing for remote assistance either as a troubleshooting or as a help mechanism. Given the current business environment where there is a need to iterate rapidly in developing client applications and testing these applications in complex production environments that are either on customer premises or hosted online in the cloud, there is a need for tools and framework such as ours.

We expect to continue investigating ways to further improve this tool and enhance its functionality. In addition we will look at how this tool and its variations can be adopted by other product teams as well as how other product teams employ or implement other mechanisms to develop and test client applications for the Silverlight sandbox environment and other sandbox environments, the cost savings and productivity enhancements that can be realized and how these applications can be rapidly iterated upon within production environments.

REFERENCES

Wikipedia. "Sandbox (computer security)," http://en.wikipedia.org/wiki/Sandbox_(computer_security) (accessed August, 2011)
Wikipedia. "Eating your own dog food," http://en.wikipedia.org/wiki/Dogfooding (accessed August, 2011)
Microsoft. "Silverlight.NET," http://silverlight.net (accessed August, 2011)
Microsoft. "Making a Service Available Across Domain Boundaries," http://msdn.microsoft.com/en-us/library/cc197955(v=VS.95).aspx (accessed August, 2011)
Microsoft. ".NET Framework Class Library for Silverlight," http://msdn.microsoft.com/en-us/library/cc838194(v=vs.95).aspx (accessed August, 2011)
Microsoft. "Lync 2010," http://office.microsoft.com/en-us/lync/ (accessed August, 2011)
Microsoft. "Welcome to Microsoft Lync Web App," http://office.microsoft.com/en-us/communicator-help/welcome-to-microsoft-lync-web-app-HA101908015.aspx (accessed August, 2011)
Microsoft. "Reference Topology With High Availability and a Single Data Center," http://technet.microsoft.com/en-us/library/gg425939.aspx (accessed August, 2011)
Fiddler. "Introducing Fiddler," http://www.fiddler2.com/fiddler2/; http://www.fiddler2.com/Fiddler/Core/ (accessed August, 2011)
Middleman. "Middleman filtering proxy server," http://middle-man.sourceforge.net/ (accessed August, 2011)
Microsoft. ".NET Remoting Overview," http://msdn.microsoft.com/en-us/library/kwdt6w2k(v=VS.71).aspx (accessed August, 2011)
JSON. "Introducing JSON," http://json.org/ (accessed August, 2011)
Wikipedia. "Fuzz Testing," http://en.wikipedia.org/wiki/Fuzz_testing (accessed August, 2011)