Occupied Cross Platform Testing

Shilpa Ranganathan
Email: shilpa.ranganathan@microsoft.com

Julio Lins
Email: julio.lins@microsoft.com

Abstract

Are you confronted with the issue of supporting an application on various platforms? Is your test matrix increasing with each release to accommodate various versions of your platform? Teams that test multi-platform products face the choice of porting tests over for each supported platform or trying to create tests that can be shared across platforms.

This paper describes our strategy and experience for platform agnostic test development across the last two releases of the Microsoft Lync product. Our proposal consists of a methodology and design for producing test content that is adaptable to application variations, allowing tests to be created once and run for multiple platforms.

Biography

Shilpa Ranganathan is a Senior Test Lead at Microsoft. She has worked in the Unified Communications space for eight years. She is passionate about working on innovative techniques to improve software testing and helping create and ship high quality products.

Julio Lins is a Senior Test Engineer at Microsoft, working on Unified Communications technologies. With a background in development of large scale corporate systems and a B.S. in Computer from UFPE in Brazil, Julio joined Microsoft 6 years ago into the Unified Communications group. Julio has been focusing on improving the test process through automation, new testing tools and techniques.
1. Introduction

Microsoft Lync is an enterprise unified communications platform. With Lync, users can keep track of their contacts’ availability; send an instance message; start or join an audio-video conversation or web conference; share your desktop; or make a phone call. It replaces a table top phone with a much more complete experience across a desktop, tablet and smartphone.

This paper focuses on the challenges involved in testing the Lync client software – excluding the server side – over several releases that have been including new platforms with each new development cycle.

1.1 What is driving Lync cross-platform testing?

Microsoft Lync currently supports 13 different client platforms or variations, as well as multiple released versions. We have released a desktop application for Windows; a managed SDK API for both .Net and Silverlight that allows other companies to implement their own Lync client; a web client; and mobile clients for iPhone and Windows Phone 7.

The number of platforms we need to support is still on the rise given the market trend of platform diversification, especially in the mobile markets.

This trend drives the need for increased efficiencies around code reuse. These clients will also need to interact with each other and their multiple versions. This is the key motivator for us to investigate and invest in a strategy for cross platform testing in the Microsoft Lync team.

1.2 What is our background?

To help us identify a testable unit among the many variations of Lync clients and their platforms, we defined the concept of an endpoint, meaning a given client running on a given platform. For example, the test scenario of an instance message conversation involves two endpoints. They could be of the same type, such as two Lync desktop clients; or of different endpoint types, such as a Lync desktop client exchanging instant messages with a Lync iPhone client.

The endpoint concept may also be extended to a specific entry point to a given client. For example, the Lync desktop client being exercised through its user interfaces is one endpoint type, whereas the same client, when exercised through its public SDK, represents yet another endpoint type. This is defined as such because the implementation of the interaction with the product in each case requires different testing mechanisms.

The organization structure we follow in the Lync team has separate product teams dedicated to developing and maintaining each different Lync client. For example, the Lync Windows desktop client is developed and maintained by one team; different from the team working on the Lync client for iPhone, for instance. This goes as far as separate teams for developing and testing a specific endpoint type, such as the public SDK for the Lync desktop client.

An interesting consequence of this team division is that a given feature, say instant message, is tested by team-A for client-A, and by team-B for client-B. This is mostly due to the platform-specific knowledge required for developing and testing on each platform. In fact, each Lync client, or even endpoint type, is a separate product, with product code sharing analyzed on a case by case basis.

Features are made available on each endpoint type as the maturity of the product evolves. In the past, we chose to implement test cases and test abstractions that were client/platform specific. This gave us more flexibility and little need of coordination between feature owners as they did not have to rely on each other’s tests. Given that each client had its own release schedule and set of priorities, using different test collateral gave the testers the flexibility to modify the tests as they chose fit to adapt to their schedules. Each test team implemented and maintained the same test scenarios across their respective Lync clients.
One side effect is that test scenarios were written against a test library whose signature was tightly coupled with the underlying platform layer code, making it only usable for that particular endpoint type. This meant that even the high level test steps, required for verifying scenarios such as the instance message conversation, were rewritten for each endpoint type.

Changes in the definition of common scenarios caused all the test owners to change their tests. As the product code evolved and was refactored, nearly independently for each platform, test teams had to create the test collateral from scratch for each release.

This meant that the size of each team would have to expand to accommodate this rewriting of the existing tests plus the development of new scenarios. This impacted our ability to support multiple platforms with the available resources and time. It also resulted in considerable duplicate efforts across the Lync organization.

As new Lync endpoint types are added with each release, coming up with a way to reuse the same test scenarios across different platforms and versions became a need and the possible return on this investment became much higher.

1.3 What is our strategy?

The above issues led to the following strategy: reusing a test case across different platforms needs to start during the test planning stage. If a test case is created with only one particular platform or endpoint type in mind, it will tend to make assumptions that do not apply to other platforms. Thus it may be cheaper, initially, to create a new copy of the same test case rather than to modify the platform specific one. The process that we outlined to help ensure we designed our tests to be cross-platform consists of the following steps:

Step 1: Form an organization of cross-platform feature test owners

Form a team of feature test owners of the same feature across platforms. One representative from each platform is ideal. This feature team is responsible for identifying the set of shared test cases for that feature. They are also responsible for implementing the tests and ensuring they work across platforms.

Step 2: Identify tests that should be shared

Not all tests should be shared. There are some tests that are needed to validate the intricacies of each platform; clearly these cannot be shared across platforms. For others, the ROI of sharing is not that high across platforms. Identifying a list of shared tests is the responsibility of the cross platform feature test team.

Step 3: Design and Implement shared tests

Once the sharable tests are identified and agreed upon by the cross platform feature test team, the team designs common interfaces and test code on top of those interfaces. They identify owners within themselves to implement an initial set of common tests. This helps to ensure that there is consistency in how the tests are implemented and ensures that all platform owners are aware of how the test is implemented.

1.4 Design Principles and Infrastructure to support shared tests

Our goal was to create a unified set of tests, test libraries and a framework, so that the same set of tests can target all Lync platforms. The high level strategy that we adopted was:

a) Create a framework to expose interfaces that will be based on the scenarios that a user can accomplish using a Lync client, such as, “Make a call”.
b) Create tests that use the functionality exposed by the “interoperability interfaces” (from step a) rather than use the underlying product testing mechanism directly (which is endpoint specific).

c) For each different endpoint type, there may be an adaptor layer necessary to translate from the “interoperability interfaces” to the actual endpoint implementation in case the interoperability interface is incompatible with the underlying object model.

1.5 Verification mechanisms

Sharing test code requires a conscious effort, as it demands constant policing to avoid duplication of code, or, even worse, common code bound to a specific platform. Hence we put the following verification mechanisms in place to ensure that the teams were following the process and design principles outlined.

**Code reviews** of the shared tests, interoperability interfaces and adaptors were performed by the cross platform team to ensure the tests meet requirements for all platforms. They validate if shared test cases are following the identified design principles, hence avoiding any interaction with platform-specific components or APIs (object models).

Shared test cases are also not allowed to make any assumptions of endpoint specific behaviors, such as types of users, platform-specific feature behavior (unless abstracted by the generic interface), phone numbers, etc.

Each shared test has a set of tags to indicate which platforms they have been verified to have passed for. Not all product versions across platforms might have the exact same feature set. Even though a test case is multi-platform, it does not mean that the case will run for every platform. The test case shall run for the platforms which the test scenario is valid for.

Test owners have the responsibility to identify which tests are run on which platforms and will tag the tests accordingly once they have validated that test works for their platform of responsibility. This helps when the test needs to be modified. Once modified the test should be re-validated on all supported platforms, as indicated by the tags, before the check-in.

2. Implementation overview

The main purpose of the shared code design is facilitating the reuse of a set of common scenario interfaces across different products, while keeping the code isolated from platform particularities.

Microsoft Lync testing involves a vastly different set of platforms, among them Windows desktop, Windows RT, iOS, Android and Windows Phone. Since not all those platforms have considerable commonalities, we chose an approach that allows us to run test cases on the desktop and interact with the product applications running on remote devices.

While it could be feasible to write the scenario driver portion on the device itself, it would require a common language (perhaps plain C++; or a script language) and platform adaptors so that the test case could be triggered (run) on all platforms.

So the main advantage of running test cases from the desktop is that the test cases can be written in C# and run on a Windows desktop, making full use of the .Net platform, while each endpoint wrapper is written specifically for each platform, hence making full use of its host platform. What makes the communication possible between the test case and an endpoint is a communication channel; which has been implemented by either .Net Remoting; or a simple in-house XML based protocol for remote procedure call in test cases.

Another logical reason for running the test case outside of the device is that most Lync scenarios involve multiple endpoints. For example, exchanging an instant message involves two users, each signed into
their own device. So the support necessary to coordinate multiple endpoints would have to have been implemented for all platforms.

Before exploring the in-depth design of our test framework, it is necessary to establish a set of requirements imposed by multi-platform testing.

2.1 Multi-platform testing requirements

Our strategy of multi-platform testing focuses on sharing reusable units across the different Lync endpoints. The items listed below constitute reusable units that we intended to share:

- **Full test cases** that verify common scenarios across platforms. Examples: send and receive an instant message, make an audio call, change availability, etc.

- **Partial scenarios** that are used by shared test cases. Examples: send an instant message, receive an instant message, start an audio call, accept an incoming audio call, etc.

Thus, the requirements are intended to make it possible to reuse both full and partial test scenarios across multiple products. A few requirements should thus be enforced:

- **Test cases and partial scenarios are isolated from implementation details**: each test should be agnostic of implementation details or APIs specific to a given platform. This can be enforced by the namespaces that the test case code is allowed to access.

- **Configuration details are external to the test case code**: test setup or configurations pertaining to specific platforms should be implemented outside of the scope of the test case code.

3. Framework elements

The design of the framework has five main elements, listed below and shown on Figures 1 and 2.

1. **Single-Client Library (SCL)**: defines a set of interfaces that are generic enough to represent any Lync client object model. The interfaces are implemented for each client on top of their Test APIs through an adaptor or directly implemented by the test APIs.

2. **Multi-client library (MCL)**: implements scenarios on top of the SCL that involve multiple endpoints, for example, establishing an instant message conversation between clients A and B. Since the scenarios are implemented on top of the SCL, the MCL can be used on any client platform.

3. **Test API**: implements an encapsulation of the actual product API or GUI that is exposed for testing. Usually abstracts some of the complexity of the underlying model, such as an asynchronous API or UI automation framework. The Test API implements the SCL interfaces directly or through an adapter.

4. **Endpoint manager**: allows a test case to create the required endpoints without knowing about implementation details of those endpoints. All the information necessary to connect the SCL interfaces with the implementation is metadata external to the test case.

5. **Service interfaces (SI)**: represents concepts such as IFileSystem (exposing a generic interface to interact with file systems that could be different for different platforms) or IRegistry (exposing a generic interface to interact with registry) that offer a standard way for test cases to execute their setup requirements. These services may or may not be available for a given platform.
Figure 1 depicts the relationship between the test case and the two components that build up complete reusable scenarios: the SCL and the MCL. The SCL is used directly by the test case when it needs to make atomic actions or verifications that involve a single endpoint, such as Sign-In. The MCL is used for composing actions into a small or complete scenario.

The MCL implements a set of highly reusable scenarios implemented on top of the SCL, hence they are independent of endpoint type. The MCL provides a way for a test case to reuse at least part of the scenario of a similar test case.

A typical use of the MCL involves interacting with multiple endpoints to achieve a scenario. For example, to establish an audio call, it is necessary to make a call from endpoint A, accept on endpoint B and verify each endpoint is connected after the call was accepted. So, one can visualize the MCL as an aggregation of SCL actions and verifications.

A test case may use one or more MCL methods to achieve the scenario it needs to verify. The MCL should contain whatever part of test cases that have a good chance of being reused by other test cases. For example, consider the test cases Establish-Audio-Call and Hold-Retrieve-Audio-Call. Both test cases need to establish an audio call. Moving the establish call scenario to the MCL allows more complex test cases to reuse the same test steps for any endpoint type.

Figure 2 shows the role of the endpoint manager and the service interfaces. The test case requests an endpoint to the endpoint manager which, in turn, uses the service interfaces hosted on a given machine to create the endpoint. The number of endpoints spawned on each machine depends on constraints imposed by the endpoint type. For example, a mobile device will usually allow only one endpoint per device.

The actual endpoint type to be used for the scenario execution is specified in the XML configuration. The configuration will contain all the necessary dependencies for the test run, such as device types, number of devices, binaries, supporting files, all of which are consumed by the test harness in the process of automatically setting up the test execution environment.
The configuration also contains the definition of the endpoint type to be used. The definition is essentially a pointer to an endpoint factory which knows how to start the endpoint using the services offered by the Service Interfaces. The Endpoint Manager loads the factory as a plugin and delegates to it the task of creating and disposing endpoints.

The Endpoint Manager allows test cases to request the creation and disposal of endpoints without actually knowing the type of the endpoint being used. This allows the same Establish-Audio-Call test case to be used for two Lync Desktop clients as well as for two Lync Windows-8-Style clients (former Metro style clients).

The test case may also need to use the service interfaces to achieve its scenario, such as disconnecting the endpoint network while a user is signed-in. Such use of the service interfaces directly from the test case will limit the endpoint types which the test case is compatible with, since not all platforms will offer the same services. Some tests do directly use service interfaces, despite this limitation.

4. Lessons learned

While reusing tests across platforms increases the return on investment as we support additional platforms, supporting cross-platform tests increases the cost of test development. In this section we describe pitfalls, lessons learned and some best practices we put in place to help mitigate some of the pain points.

4.1 Standardization of APIs across products

The similarity of the product APIs across platforms has a direct impact on the amount of effort necessary to establish a common set of shared tests. Products that expose a standard API for testing across platforms offer a higher return of investment for sharing tests.

It is important to note that if the product itself follows a layered architecture and shares code among the platforms on which it runs, it will more likely expose similar APIs across platforms, hence it will positively impact on the reuse of test code.

When the product APIs are similar between platforms, it will be easier to define common interfaces for the same functionality across platforms. Whereas, when the differences are great, adaptors might be needed for connecting between very different APIs. Another drawback of diverse product APIs is that arriving at a common set of test interfaces will require more effort from the feature test team in their pursuit of agreement on what the common code should look like.

For future releases we are investigating the feasibility of standardizing the application layer interfaces for the different client types, in an attempt to have a similar API set exposed for testing for all clients.

4.2 The engineering infrastructure impact

The engineering aspects of the company greatly impacted the feasibility of sharing test code.

Our experience in testing Lync indicates that it will be more productive to share test scenarios among products that share the same source code management system. In recent development cycles, our product source code was distributed among three source code management systems. The end result was that products sharing the same source code management systems would live up to its full test sharing potential, whereas products on different source code management systems would have partial sharing, dependent on other variables as follows.

The build process and tools influenced the cost of sharing. In the ideal scenario, all products and test code are built from the same source control system and tests are executed upon successful builds.
Breaks are detected as soon as they are introduced. In circumstances where the product and its tests are built and maintained by separate teams in separate source control systems, breaks in functionality that were introduced by one team will take longer to be detected and hence impact the productivity of all teams involved.

Finally, the ability of triggering test runs for all platforms impacted the sharing process. In the ideal scenario, a tester of a given platform is able to kick off test runs of the same scenario for other platforms with minimal test setup efforts. This allows changes to be verified across all target platforms. The more complicated the test setup for a given platform is, the greater will be the cost of validating changes for that platform. This will ultimately impact on the team’s ability to verify product changes when product code is shared across platforms.

We are pursuing conversations with the various teams to try and consolidate the source code management systems for the different clients. While some teams are agreeing to this we will still have some teams using different source code systems at least for the next release and we will continue to pursue this discussion with them.

4.3 Schedule

When different teams are working on each platform, differences in schedules and priorities also greatly contribute towards the success or failure of sharing tests. Even though teams may intend to share tests, one team may be further along in the product development cycle, and hence may have a better idea of the feature set to implement and test. In order to conform to their schedule needs, they may have to go ahead in developing the tests without much knowledge of the requirements from other teams who are not as far along in the product development cycle. This may lead to developing tests that might not fit the need of all platforms. In our experience, teams with a stable test base may be averse to considerable refactorings, or any at all, whereas teams joining a common test code base will demand adjustments in interfaces or partial scenarios when they differ much from their product behavior. This conflict of interest in the common code base may result in new teams only partially adopting the shared tests. Hence, differences in schedules may adversely impact the sharing process, possibly reducing its scope.

For example, the Lync desktop client was the first to introduce the feature to share the desktop. At the time this feature was introduced, no other client was ready to implement this feature, so the test owner designed and implemented the common interfaces and tests based on the feature requirements for the desktop client. We had around 20 tests stabilized and running successfully, when the decision was made to add this same feature in the managed client platform SDK. We found that modifications were necessary to the existing tests, as the way to initiate desktop sharing between the 2 clients was considerably different. One client exposed a single API to initiate the sharing, whereas on the other client the initiation was split into 2 API calls. This basic change impacted all the tests. Clearly the desktop client team had reservations around changes to these tests as they now had to modify all tests as well as validate all the tests to ensure no regressions. The outcome of this conflict was that we introduced a new interface to expose the sharing initiation. This change was made by the managed client platform SDK owner and then both teams had to execute the tests based on these changes against their platforms. After consecutive bug fixes, we were finally able to stabilize the tests for both platforms.

Schedules are based on business and market trends so we will continue to face this problem.

4.4 When full tests cannot be shared, share smaller scenarios

It was our experience that not all test scenarios were exactly identical among platform specific versions of our product. The feature set is not an exact match across the board. Features may be restricted to a given platform set of capabilities. This is also aggravated when the underlying product object model differs between platforms.
While each platform may have a specific set of features, there are likely to be common scenarios that are shared across platforms. For example, all Lync clients share the Sign-In and Instant Message features, but, at the time of this writing, not all clients offer the VoIP call feature.

One interesting aspect of the approach we took is that the common interfaces (SCL) have a generic reference (object type in C#) to their underlying implementation. Although this may seem odd from a common interface design, it allows test cases to access the underlying implementation, hence its platform specific actions, when needed. The use of an object type property instead of casting is intended to enforce the support to the cut-through concept.

This is useful for platform specific scenarios that start from a common scenario. It allows a platform specific test case to reuse the code necessary for achieving the common scenario, then cut through the interface layer to access the platform specific code. For example, the steps involved in testing a VoIP call include signing-in and creating a conversation, both of which are scenarios that can be reused across all platforms.

It becomes essential to not only identify reusable scenarios, but to also define which parts of these scenarios will be platform specific and which parts can be reused across platforms. Hence, the role of the cross-platform feature team is not just defining the ideal reusable scenarios, but also focusing on the most return of investment for tests shared across platforms.

4.5 Latent costs associated with sharing tests across platforms

The reuse of tests adds additional costs to the automation development. Although we are unable to provide a formula for the return of investment in test reuse, this section describes some situations in which we observed additional test development costs.

Differences in development environments will delay the development and verification of common test code. Sometimes one test owner does not have the specific environment to build/test on a particular platform. In this case he/she has to wait for that environment to be setup or ask someone else to run the test on other platforms.

The coordination among the feature teams across platforms to define the shared test cases and create a test design to accommodate all platforms adds more time to the process. For example, the code review process for a change in common code may take longer as all stake holders, in our case from different teams, need to review and approve it.

The stabilization of the tests on the various platforms is certainly time-consuming. Although each test scenario is not replicated for each platform, it still needs to be stabilized on the target platforms. We found that once a test has been stabilized on one platform it takes approximately one forth the amount of time to stabilize on new platforms.

We are working closely with our leadership team to ensure that these costs are accounted for during the planning stage so that we are better prepared.

5. Conclusion

Reusing tests is not exactly a straight-forward process. While one may likely provide very strong arguments for the reuse of product code, test automation code is one of the tools for asserting the product quality. The reuse of test is a trade-off between the cost of development and the final impact on the product quality, often measured by the amount of bugs in product code found by automation.

The amount of code to be shared will be impacted by the variables mentioned in the previous section, involving the engineering aspects, such as source code location, as well as organizational aspects, such as team schedules and the disparity of the product characteristics across platforms.
As the title of this article jokes with, cross platform testing will very likely bring up special cases with impracticable reuse costs. A wise test organization will look to trade-off the amount of tests reused among platforms, balancing between the savings of reusing code with the need for platform-specific testing, given by platform-specific features or prohibitive reuse costs.

This article presented some guidelines and a framework to be carefully analyzed by a test team working on a multi-platform product, and intending to save on test development costs by reusing test scenarios.

In the Microsoft Lync team, we are continuing to expand our investments in the engineering infrastructure to create a common framework to support all platforms. Even though we are unable to measure precisely the return of investment in reusing test code, our experience indicates this is a vastly improved process and that the Lync test team has embraced it. Having seen it at work, we can't imagine going back to a world where we replicate all test efforts for each new platform.