UNIVERSITY OF TROMSØ UIT

FACULTY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Javza

A runtime supporting dynamic app configuration and integration in asymmetric systems

Anders Tungeland Gjerdrum

INF-3981
Master's Thesis in Computer Science
July, 2012

Abstract

The advent of cloud computing alongside with pervasive form factors such as smart devices, introduces a new meaning to asymmetric system models. These new clients act as a presentational layer alleviating much of the computational and storage concerns to cloud services. The application platforms associated with these creates new opportunities for third party developers to provide domain specific applications (apps) to smart devices. Restricted interaction surfaces on smart device introduce new challenges to how apps are managed on these. Moreover, for security purposes as well as strict resource requirements, apps running in these environments are commonly subject to very strict isolation.

We conjure that there are benefits with allowing automatic configuration apps onto a client system. Furthermore, we suggest that integration between apps residing at the same host, as well as different hosts, is beneficial to system functionality. By enabling this we could alleviate much of the interaction necessary for users and reduce the time consumed in using such smart devices.

The concrete asymmetric system explored in this thesis is the Windows 8 app platform. This platform poses several hindrances to our conjecture. For security reasons, automatic configuration of apps onto a client system is prohibited. Apps run inside a sandboxed environment where they are isolated from the system and other apps. Access to resources is prohibited unless explicitly allowed by the user. As a consequence, communication between apps is forbidden. This is further complicated by the fact that apps in this environment are suspended when not in use.

This thesis introduces Javza, a runtime to support dynamic app configuration and integration in Windows 8. Javza provides support for automatic installment of apps based on simplified contextual information. Furthermore, it provides app integration by allowing apps to share data, both within the same and across different systems.

We present and evaluate the associated performance costs of deploying Javza inside a Windows 8 environment. We further evaluate the applicability of Javza by implementing a specific use case involving collaborative search. Lastly we discuss some of the security implications associated with our design, and some future improvements in Windows 8 to support our conjecture.

Acknowledgements

I would like to thank my advisor Prof. Dag Johansen for all your guidance and support. Your enthusiasm is highly infectious.

I would also like to extend my gratitude towards my co-advisor Krister Mikalsen, your perspective and insight has been much appreciated.

Further, I would like to thank my Girlfriend for listening to my mad rambles these last 5 months. Your compassion knows no boundary, I love you.

I would like to thank Robert Pettersen for his valued friendship and helpful discussions in the last five years. I would like to thank my family, especially my mother and father, for being true sources of inspiration.

Lastly, I would like to thank my fellow colleagues at the IAD research lab, including Audun Nordahl, Åge Kvalnes and Dr. Håvard D. Johansen for their insight and help throughout this process.

Table of Figures

Table of Content

List of Acronyms

XML = Extensible Markup Language
IPC = Inter-Process Communication
DHT = Distributed Hash Table
GRS = Group Recommendation Service
AMC = App Management Component
IDC = International Data Corporation
CPU = Central Processing Unit
API = Application Programming Interface
GPS = Global Positioning System
ARM = Advanced RISC Machine
RISC = Reduced Instruction Set Computing
OEM = Original Equipment Manufacturer
GUI = Graphical User Interface
TCP = Transmission Control Protocol
IP = Internet Protocol
NIC = Network Interface Card
COM = Component Object Model
CLR = Common Language Runtime
OS = Operating System
I/O = Input/Output
TPL = Task Parallel Library
LINQ = Language Integrated Query
JSON = JavaScript Object Notation
VM = Virtual Machine
JVM = Java Virtual Machine
W3C = World Wide Web Consortium
UDDI = Universal Description Discovery and Integration
JIT = Just-In-Time
WNS = Windows Notification Service

1 Introduction

The advent of cloud computing has introduced a new paradigm in systems design [1]. Cloud providers deliver computational power and storage as a service to consumers at different levels of abstraction.

Infrastructure as a Service (IaaS), delivers raw computing power and storage to consumers in form of virtual machines with networking capabilities attached and block data storage. Platform as a Service (PaaS) provides system software for building applications in the cloud. This includes runtimes for these to execute in and structured storage capabilities such as databases. At the very top end of the abstraction is Software as a Service (SaaS), providing complete application services accessed through web browsers or other light clients.

Enterprises are becoming big consumers of cloud technology. CISCOS Global cloud networking survey [2] asked 1300 important IT decision-makers worldwide about their companies cloud migration strategy. A large percentage of these answered that they have, or at least are planning to migrate much of their corporate infrastructure into the cloud.

Private users also follow this trend and are becoming increasingly dependent on the cloud. The cloud provides services including everything from email and social networking services, to productivity and collaborative tools. All of which are directly consumable by end users. The clients of these cloud services are becoming increasingly pervasive and are no longer restricted to desktop computers. They now include more light weight form factors such as smart phones and tablets. International Data Corporation (IDC) [3] reports that nearly a billion such smart devices where shipped in 2011, and shipments expect to double by 2016. More and more consumers now possess multiple smart connected devices, for different purposes. We refer to this new client-server model as an asymmetric systems model.

In this model, computation and storage are asymmetrically distributed between the client and the server. Light weight clients act as a presentational layer of the system, while the server provides the storage and computational power. Smart devices act as these light clients which then consume cloud services, bringing content and functionality to the devices.

With the introduction of these light weight form factors come new application platforms delivering third party application support. These provide generic and open APIs for creating apps specifically targeted towards these smart devices.

These apps are typically very restrictive in functionality and domain. They focus on the presentational aspects of the system, pushing many the storage and computational concerns to cloud services. These could either be platform provided service accessible from all platform apps or third party services associated with a particular app.

Smart devices typically have a small user interaction surface. However, users must adhere to the same approach to application management designed for desktop computers with much less restricted interaction capabilities.

To acquire apps, users are required to lookup, download, validate and install each of these onto their device explicitly. Some of these are only necessary in special cases and under certain conditions. The applications that aren't in use, hog up space, and even resources on the devices. Other application forms such as web application execute through the browser in mostly any environment without installation. Smart device apps are often subject to the same restrictions as web applications, but still require a rigorous installation procedure.

Specifically, these restrictions limit the app's capability to interact with other apps. Consider a scenario where a set of apps with different usage domains are installed on the same smart device system. They are being used by a single individual as a part of achieving a goal, satisfying different parts of it. These apps should be able to share data in order to contribute to each other's fulfillment of the goal. Also, consider if the individual is contributing to a social group which is cooperating in achieving this goal, these apps again should be able to share state to more effectively perform their joint tasks.

App platforms often restrict this kind of behavior by isolating apps from one another and the system for security and resource considerations. Other apps that could benefit from using this data in their own domain aren't able to do so. In order for the scenario to have the same effect, the user must then manually update the state of each application as the work progresses.

Building upon the arguments earlier made with regards to restricted user interaction surfaces, it is not difficult to observe that this approach is inefficient.

1.1 Thesis Statement

This thesis shall develop and evaluate a runtime system for dynamic application configuration in a concrete asymmetric system environment (Windows 8). This run-time must support integration across applications on the same host, as well as applications receding on different hosts. Alternative solutions with and without server-support must be explored.

The prototype will evaluate important aspects of the prototypes performance. Furthermore we will evaluate the prototype by implementing a use case, more specifically collaborative search. The evaluation will include suggestions for further optimizations and extensions, and possible implications for adopting these.

1.2 Scope and Assumptions

The underlying platform of interest to our thesis is the Windows 8 application platform. We conjecture that asymmetric system models will benefit from allowing dynamic and automatic application management and integration, both across and inside asymmetric resource constrained environments. We will implement a prototype for dynamic application management and integration. Our focus in this thesis is on alternative approaches to app communication across different hosts without server support. We will evaluate the possibilities and limitations of our prototype in the context of the underlying system. Furthermore we will illustrate our design in a broader context, by implementing a use case involving a collaborative search experience. The two major functional requirements our prototype must fulfill are the following:

- The prototype must support communication between apps located inside a single client, as well as across different clients to support integration amongst these.
- The prototype must support automatic configuration of an app environment onto the client system.

1.3 Method and Approach

The final report of the Task Force on the Core of Computing Sciences [4], divides the discipline of Computing into three paradigms:

- *Theory* is rooted in mathematics and includes four steps in the development of a coherent and valid theory. (1) First one characterizes the objects of study. (2) Then hypothesize possible relationships between these (theorem). (3) Following this one seeks out to falsify or prove the relationship. (4) Lastly the result is interpreted.
- *Abstraction* is rooted in the experimental sciences, and follows four steps to research a phenomenon. (1) One forms a hypothesis relating to the phenomenon. (2) Then one creates a model based on this and makes predictions. (3) Following this an experiment is constructed and the resulting data is collected. (4) Lastly the results are analyzed.
- *Design* is rooted in engineering, and follows four steps in the process of constructing a system. (1) One states the requirements and (2) specifications of the system. (3) Then design and implements the system. (4) Lastly the system is tested to conform to the initial requirements and specifications.

In this thesis we utilize the design paradigm. We state the requirements and specifications of the system related to our conjecture. We present a system architecture based on these requirements and implement a prototype. The prototype system is then evaluated in a series of experiments for conformance with our initial requirements and specifications.

1.4 Outline

The outline of this thesis is as follows:

- **Chapter 2** details the notion of an asymmetrical systems model. We introduce Windows 8, the system subjected to our conjecture, and evaluate it with regards to our functional requirements.
- **Chapter 3** details the architecture of Javza, a runtime to support dynamic app configuration and integration.
- **Chapter 4** explains the implementation of Javza, evading the restrictions posed by Windows 8
- **Chapter 5** evaluates the associated performance costs of deploying Javza in Windows 8.
- **Chapter 6** presents a specific use case and evaluates the usability of Javza in this context.
- **Chapter 7** discusses some of the Security implications by adhering to our conjecture. Furthermore, we suggest some further adaptations to the Windows 8 platform and list the related work.
- **Chapter 8** summarizes this thesis and provides some concluding remarks and thoughts on future work.

2 Asymmetric System

Before we explain or prototype design, we will explain the concept of an app platform and how it relates to the asymmetric system model. We will give a brief introduction to the major components that often exist in such, often referred to as an app platform, and relate them to existing systems. We then proceed to describe the new Windows 8 app model, our subject of interest in this thesis and its major components. We conclude the chapter by looking at some properties of the Windows 8 app platform and evaluate this with respect to our initial conjecture.

2.1 App Platform

Asymmetrical system models have existed for some time [5], more recently in the form of app platforms.

In the last years this market segment has exploded in terms of size and revenue. The distinction between personal computers and smart devices is no longer binary and we see users being in possession of a multitude of computing devices of different sizes. Hardware configurations are becoming more powerful, and it is no longer uncommon to see multicore processors in mobile devices or tablets. Many configurations exist for each use case, trading portability for complexity of work.

Smart devices typically have specialized application platforms associated with them which enable development of third-party applications for the devices. These applications, often referred to as apps, are very restrictive in domain and functionality.

Such apps exist on top of several other abstractions as well. These range from low level mobile operating system platforms such as Android[1] and iOS[2] up to the ones residing inside applications themselves, such as Facebook[3], Spotify[4] and Skype[5]. The only requirement for an app platform is that the underlying system function is broad enough to provide a rich API available to developers. The collection of cloud services, and internal systems APIs should enable developers to create new apps with non-existing features. These apps then drive the innovation of the platform further by introducing functionality not originally thought of by the creators of the underlying system.

Our focus in this thesis is on the smart device abstraction, with apps and cloud services. The combination of these two is what constitutes our notion of an asymmetric systems model. Apps

[1] http://www.android.com/
[2] https://developer.apple.com/devcenter/ios/index.action
[3] http://developers.facebook.com/
[4] https://developer.spotify.com/
[5] http://developer.skype.com/

residing on resource constricted smart device clients commonly have associated cloud services. These cloud services deliver content and functionality to the app.

An overview of the components involved in a typical app platform can be seen in Figure 1. This model includes the following three components:

- The client runtime, the execution environment in which the app run.
- The cloud services that provide content and functionality to the apps.
- The digital distribution platform, or app store, which provides apps to the client.

We will describe them each in more detail in the following paragraphs.

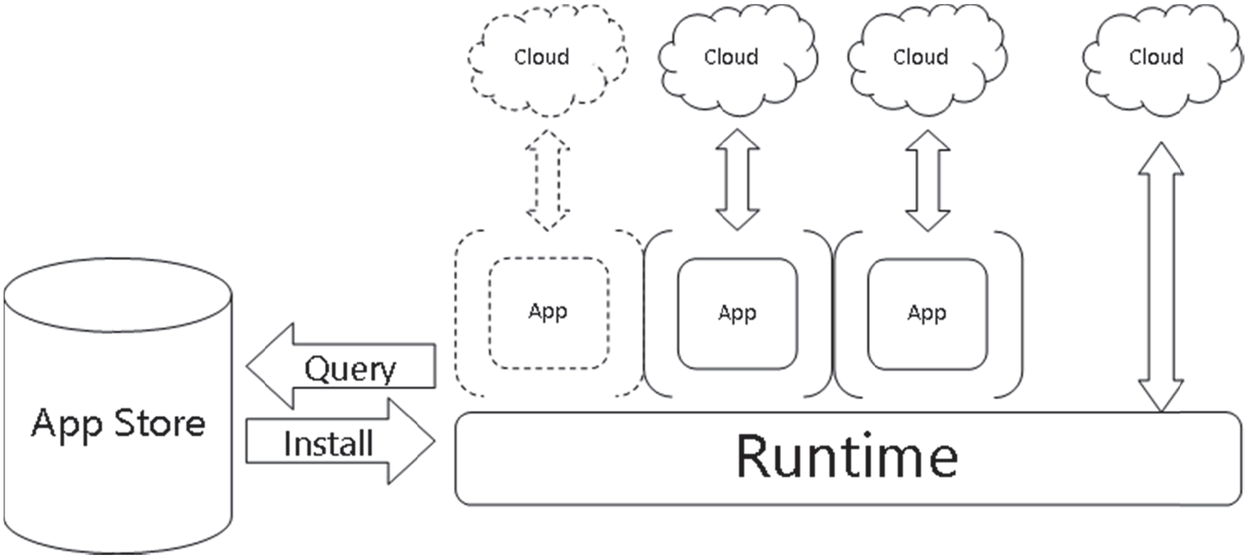


Figure 1: Application platform model

Because they run on battery power most of the time, smart devices have fairly limited power resources available. Because of this they commonly optimize towards low power consumption. Memory is limited to reduce power consumption, and generally these devices have no swapping ability for virtual memory. For the same reasons, the CPUs in these devices are much slower than laptop or desktop environments. Conserving the usage of available memory and CPU resources is therefore very important. Platforms such as the iOS or the Windows 8 platform in the next section are fairly restrictive in how they allow multitasking of apps in order to conserve memory and CPU. Also, with regards to multitasking, the screen real-estate on these devices is limited and there is only a limited display area available for use at a given time.

Cloud services pose an integral mechanism for offloading computation and data to services outside this limited resource environment. Examples of these include cloud storage and management of app data, personal data related to emails contacts and calendars. Some of these services may be provided directly to the system and through it consumable by apps. Other cloud services again are proprietary and only accessible to a particular app.

The runtime is the underlying software component that the system runs on top of with the associated system support. This component manages resources, thread scheduling and everything associated with the execution of an app. Different runtimes provide varying levels of isolation and portability. Some execute inside an interpreter engine, while other are executed inside an isolated virtual machine environment. Others yet again, are implemented in native machine language executing as common processes directly on top of the operating system. Common for them all is that they in some form provide resource management and isolation from other app instances.

Digital distribution platforms, more commonly referred to as app stores, provide a uniform and secure mechanism to distribute apps to consumers. The distribution services often require licenses to develop apps and deliver them to the store. Users must then hold accounts for purchasing and consuming apps. They generally provide a strict policy on what types of apps are allowed inside their store. Apps must adhere to the policy enforced by the store and require app certification prior to submission.

2.1.1 The Android Platform

The Android platform is the most widely deployed app platform in the world. At the first quarter of 2012 it a 59% share of the smart device market [6]. The Android operating system is deployed on a wide range of device types, including smart phones, tablets and even media center computers.

Apps developed for this platform are usually implemented in the Java programming language, but there exist SDKs for native development as well. The apps developed in Java are compiled down to byte code and executed inside the Dalvik Java Virtual Machine[6]. This provides isolation from other apps and system components.

Android apps run in a sandbox, where they are isolated from accessing system resources unless access permissions are granted by the user upon installation. The Android Operating System is built on top of a Linux based kernel. The overview architecture of the Android platform client can be seen in Figure 2.

[6] http://source.android.com/tech/dalvik/index.html

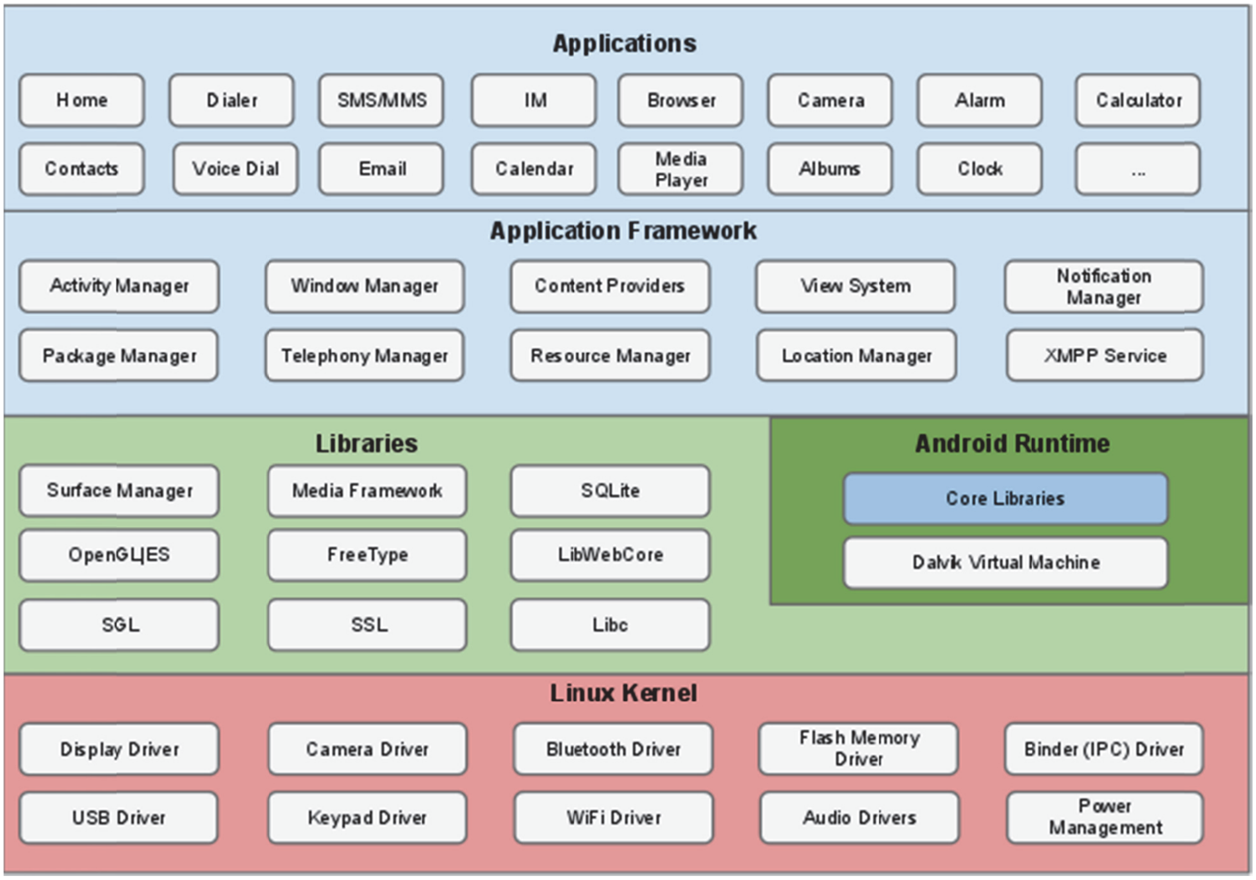


Figure 2: The Android Operating System Architecture[7]

As far as app platforms goes, Android is perhaps the least restrictive of its kind. With regards to multitasking it poses fewer restrictions on apps executing code the background.

There is however a 5-10 percent cap on the altogether consumable CPU by background processes. The OS can, when observing low memory conditions, terminate apps running in the background. This might sound strict, but other platforms enforce much harsher policies.

Unlike Windows 8, as we will see in subsection 2.2.6, Android provides functionality for apps within the same smart device to communicate. All Android components communicate using this functionality, referred to as intents. An intent message is asynchronous, contains a recipient and optionally data as well. It could be thought of as a separate component implementing a functionality which is executed upon reception, much like remote procedure calls. Intents are one to one communication constructs, and apps are not able to send system-wide broadcasts. Only the Android OS can broadcast to the entire system. The system uses intents to signal apps when important system events occur. There are two types of intents implementable in Android, implicit and explicit. Explicit intents specify the receiver of the

[7] Source: http://source.android.com/tech/security/index.html

message, and these should only be delivered to that specific app. Implicit intents can request delivery to any app as long as it implements the functionality required by the intent.

The digital distribution service for the Android platform is the Google Play store. Apps for Android devices are downloaded and installed through this distribution hub. But, unlike iOS and Windows 8, Android allows apps to be installed from outside the store, although only explicitly. Users still have to validate the app before installment. To access the store users are required to register with a Google account. This account connects all of Google's cloud services, ranging from email, cloud storage to social networks in Google+. Apps implemented for the android platform are able to consume these cloud services by approval from the user.

2.2 Windows 8

This section evaluates the asymmetric system we use in this thesis in order to validate our conjecture, the Windows 8 app platform[8]. The outline of this section is as follows:

We will first explain the general model, and the rationale behind it. Then we explain how apps in this environment consume cloud and system services. We further proceed by describing the Windows Store digital distribution platform and the requirements for installing apps onto Windows 8. We will then give a description of the underlying system, the Windows Runtime, which these new apps use. Finally, we summarize the section; evaluate the system, and how it conforms to our initial conjecture.

2.2.1 New In Windows 8

Windows 8 unlike its predecessors include a brand new app platform [7]. The system will be released in several versions, one of which will run on the ARMv7 architecture. This version, officially called Windows RT, will be available for tablet form factors. This is not to be confused with the Windows Runtime, the environment in which apps for this platform run.

Windows RT will not be released to customers as an installation package. Instead it will be preinstalled on the devices coming from the OEM partners. Each OEM is responsible for rolling out updates to its own devices. Windows RT appears to enable some form of the classic desktop environment. The package will come preconfigured with touch optimized desktop versions for both Internet Explorer and the Microsoft Office suite. On the other hand, third party developers will not be able to provide or install software for the desktop environment. Microsoft has announced that they will themselves produce a tablet based on this operating system, called Surface[9]. This tablet will come in two versions, both based on x86 and ARM. They have further announced that the next installment of the Windows Phone operating system, Windows Phone 8 [8], will build upon the same core as Windows 8.

[8] http://msdn.microsoft.com/en-us/windows

[9] http://www.microsoft.com/surface/en/us/default.aspx

The apps designed for this new app platform, are called Metro Style Apps. They are distributed exclusively through the Windows Store, and are available for both the ARM and X86/64 architectures. They get an entirely new, touch friendly, immersive interface which embraces Microsoft's new design language Metro. It is already integrated into their services ranging from Windows Phone 7 to the Xbox Media Center. The form dictates strict and clear typographic design with emphasis on the functional aspects of the user interfaces. The principal axiom of the language is to avoid diluting the visual presentation of the data with superfluous content. Figure 3 illustrates the new start screen of Windows 8 with several apps installed on the left hand side. These squares are referred to as the "tiles" of each app, and introduce a new way of showing apps that are available in the system. However, regular applications for the old familiar desktop environment are only available for x86/64 architectures.

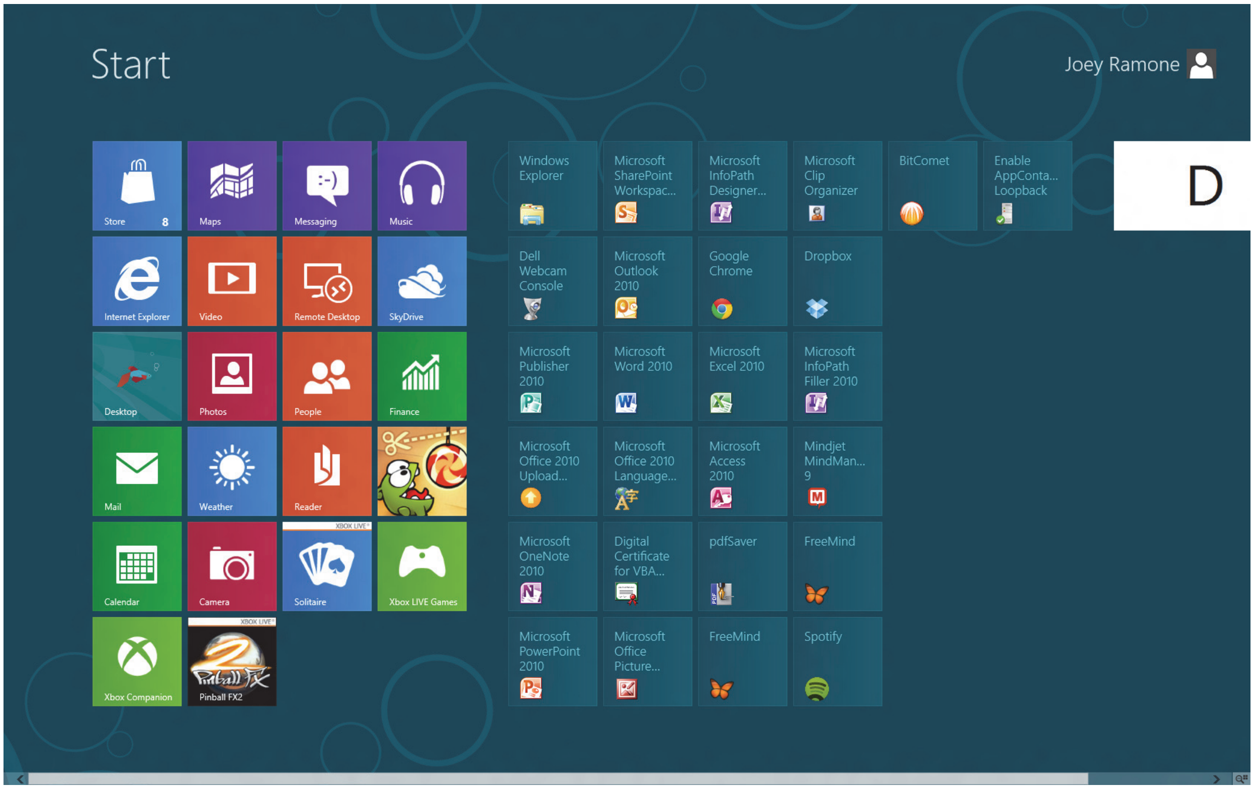


Figure 3: Metro Immersive User Interface

2.2.2 Capabilities and Contracts

Windows 8 is heavily integrated with Microsoft's own cloud services, accessible through the users Windows Live account. One important feature of this is the ability to roam apps and configuration between Windows 8 instances. Users are able to log in with their live accounts on one computer and their system configuration is accessible from other computers when they log on with the same account.

Apps are also able to consume these cloud services through the system. An example of this is being able to roam app specific data related to a single user account across system instances. To control how each app access resources, be it cloud, system or devices, each app requires a capabilities declaration.

Capabilities are explicitly declared intents to interact with the environment, embedded in the app manifest. The app manifest associated with each app contains metadata about the activation and execution of the app. These declarations are made during development and invoked during installation. They describe specific features of the system that the app requires to function, and upon installation the user is asked to approve of this. The APIs that these capabilities control, consists of an array of application layer constructs to interface with devices and system resources. Examples of which could be everything from allowing background connections and web cam capture to file system access and GPS location services.

Another class of implementable policies for apps to interact with the system is contracts. These are activatable pieces of code which are executed in the case of specific environmental events. Apps implement a specific behavior when activated in the context of these contracts. The contracts that are available for a given app is specified in the app manifest.

The search contract is an example of this. This contract is activated through the systems new search functionality which allows users to search through content inside of apps. The way in which each app chooses to implement this search contract is open. Apps can display which ever results it deems fitting in whatever way they wishes to present it. Other similar contracts include sharing between apps. In this, both apps implement the two way functionality and how the apps respond when being activated.

Common for all contracts is that they require explicit user interaction to be activated. Information about these contracts is stored in the registry upon installation alongside with the entry point and code classes to invoke the contract.

2.2.3 Installment and activation

Included in Windows 8 is a new digital distribution platform called the Windows Store. This store is the only way Metro Style Apps can be distributed to private consumers. When a developer prepares a package for submission to this store, the app manifest, together with the compiled binaries are stored in a compressed app package.

Before submission all apps must pass the Windows Store certification process, ensuring that apps conform to the stores policy. All apps are tested for package manifest compliance. This means the use of only supported APIs, runtime debugging and security validation.

Before installation, apps are further inspected by the operating system. The app package is verified for a correct signature. This signature must be made by a trusted certificate authority. In this case it is Windows Store which is trusted by all Windows 8 clients. Then the apps pre declared capabilities must be validated, first by the OS and then by the end-user explicitly. After this procedure is done the app is ready for installment.

The process from installing an app to activating it contains several steps [9]. The app manifest contains an app registration field, describing the most important aspects of the app, including the contract activation specifications. This field is used to store the metadata needed into the Windows Registry for managing the app. All apps are installed per user into the registry and there are many different registrations for different types of activations. Starting the app via for example touching a tile from the start screen activates a specific contract, as mentioned before. All apps implement this, called the launch contract.

When activated, the registry contains information to how the app is launched. The information describes whether it is an out of process activation, used in apps developed in JavaScript, or in-process activation for C# or C++. In the case of a JavaScript/HTML developed apps, the code is launched within a separate system provided binary called “wwahost.exe”.

Enterprise consumers are able to install apps outside the store by what is refers to as side loading. This involves the same strict policy for app package signing and installation as explained above.

2.2.4 Windows Runtime

Metro Style Apps run on top of a separate abstraction from regular Windows applications. This new runtime is called the Windows Runtime [10]. This should not be confused with Windows RT, the tablet version of the Windows 8 Operating System. In the ARM version of Windows 8, the ability to develop desktop applications is gone. Both the win32 APIs, and the .NET platform is unavailable to third party developers.

Windows Runtime only exposes a small subset of the services available in these. This is not a new executable runtime in the common sense, but rather a set of system-managed and maintained objects exposed by the system. An illustration of the system separation can be seen in Figure 4.

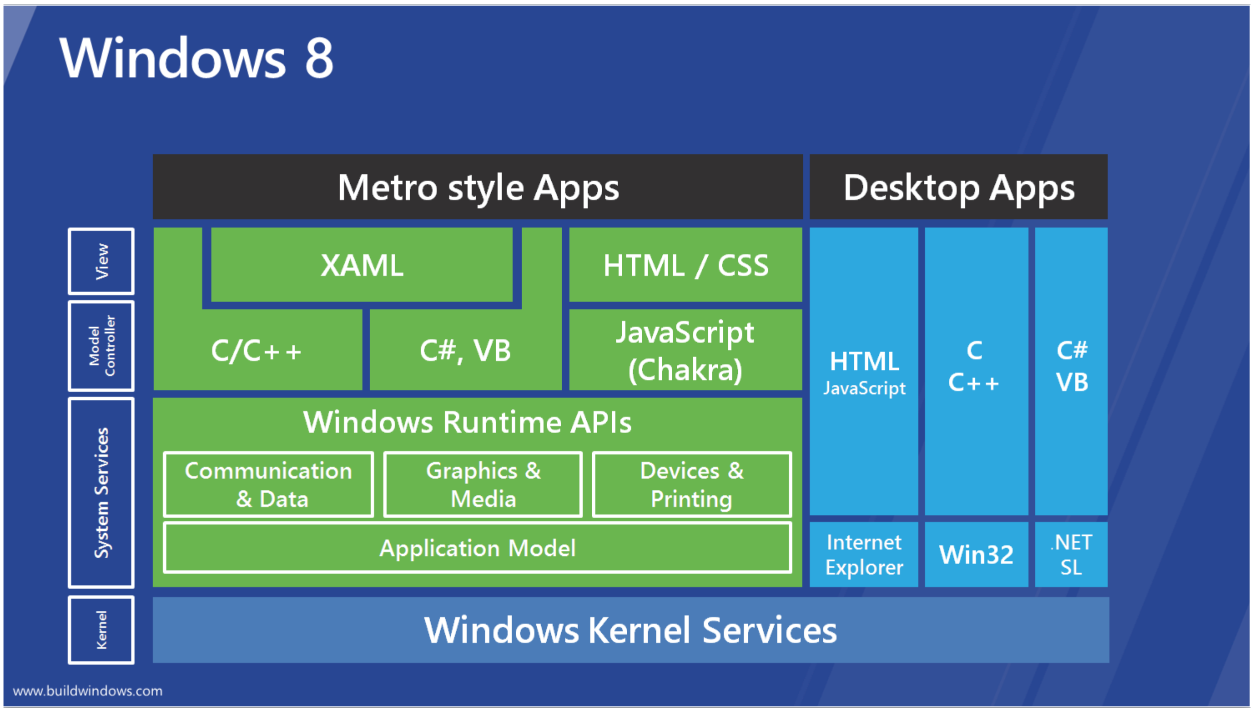


Figure 4: Windows 8 Architecture [10]

At its very core Windows Runtime is designed around an enhanced version of the Component Object Model (COM). The model restricts components from influencing the execution of other components by being self-contained and decoupled from the interface exposed to consumers of the object. It is designed explicitly with interoperability in mind and allows language neutral access to these objects.

The Windows Runtime objects are themselves built directly on top of the Windows kernel services. To enable consumption from multiple languages, Microsoft has designed language projections for each programming language to function as an adapter between COM objects and the app code. The APIs are exposed using metadata files with the same format used by the .NET framework [11]. This makes them consumable from all languages supported by the .NET framework, including native, managed and scripting languages.

An example illustration of this interaction can be seen in Figure 5. At the time of writing there currently exist projections to unmanaged app development through C++ as well as managed development through Visual Basic, C# and JavaScript. The apps developed through Visual Basic or C# code run on top of the Just-In-Time (JIT) compiler and the Common Language Runtime (CLR), just like all other .NET applications. The JavaScript language projection is run on top of the Chakra JavaScript engine, while the C++ apps are compiled into native code.

Through the Windows Runtime, there are over 800 objects available for interfacing with system resources. Windows Runtime objects are stored in the Windows registry for discovery and identification. They are exposed to consumers through entry points for activation. All libraries and apps are constructed as Windows Runtime objects and developers can themselves construct custom objects, compile them, and make them consumable across language boundaries. For example, a component written in C++ could be consumed by a JavaScript app.

All Windows Runtime objects implement two base interfaces, IUnknown and IInspectable. The IUnknown manage the existence of these through reference counting. It also enables users to get pointers to other interfaces exposed by the given object. The IIinspecable interface must be implemented in order to provide projection across languages.

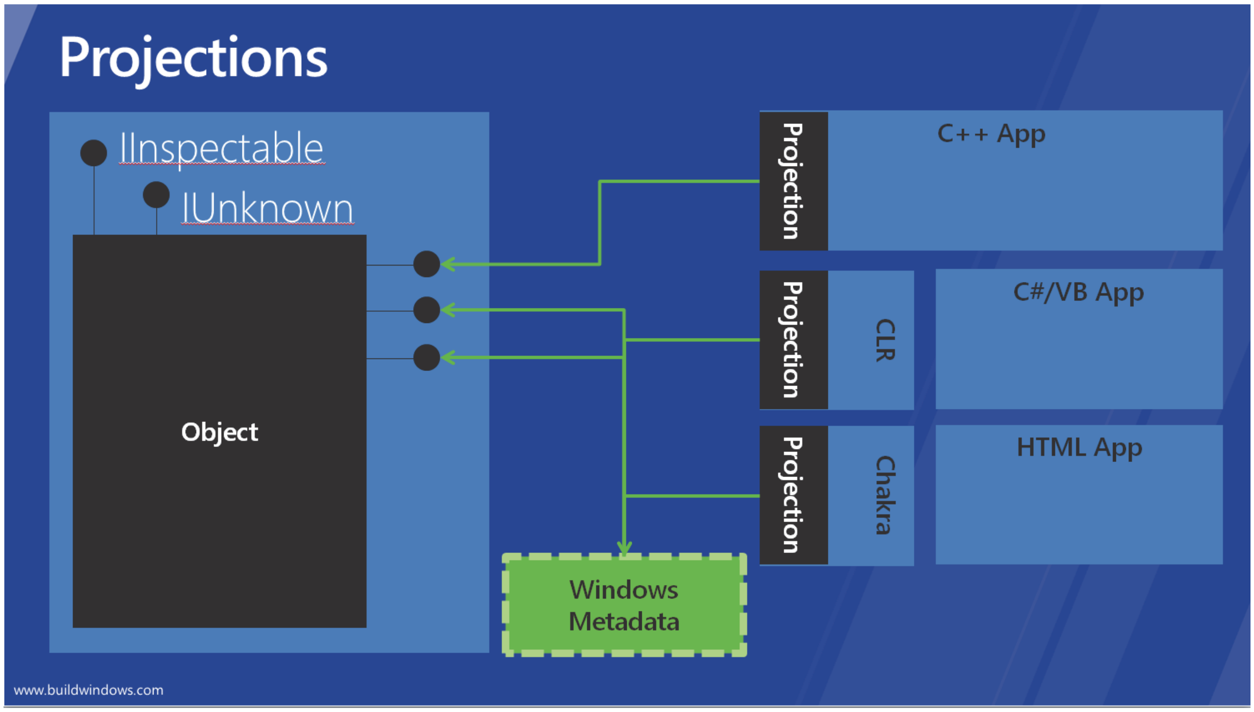


Figure 5: Windows Runtime Object Projections [10]

2.2.5 App lifecycle

Metro Style Apps are intended to run in very diverse and resource constrained environments. Windows 8 with Windows Runtime included, is designed specifically for this purpose. In comparison with the Android platform's multitasking abilities, Windows Runtime's is much more restrictive. When installed and activated, Windows 8 Metro Style Apps run in separate processes. They go through what is called an app lifecycle similar to the multitasking abilities in iOS [12].

The apps transits through several states while installed into the system, as depicted in the diagram below (Figure 6). Apps not running in full screen are suspended to conserve resources. They are notified of the event and given 5 seconds to save any state before being suspended. While suspended, these apps are present in main memory but no part of it is being scheduled by the operating system kernel for execution. In the case where the system is low on memory, these apps are eligible for termination without notice. They are then removed from memory, hence the importance of saving state when suspended. As a consequence, there is normally only one app running at a time in the system.

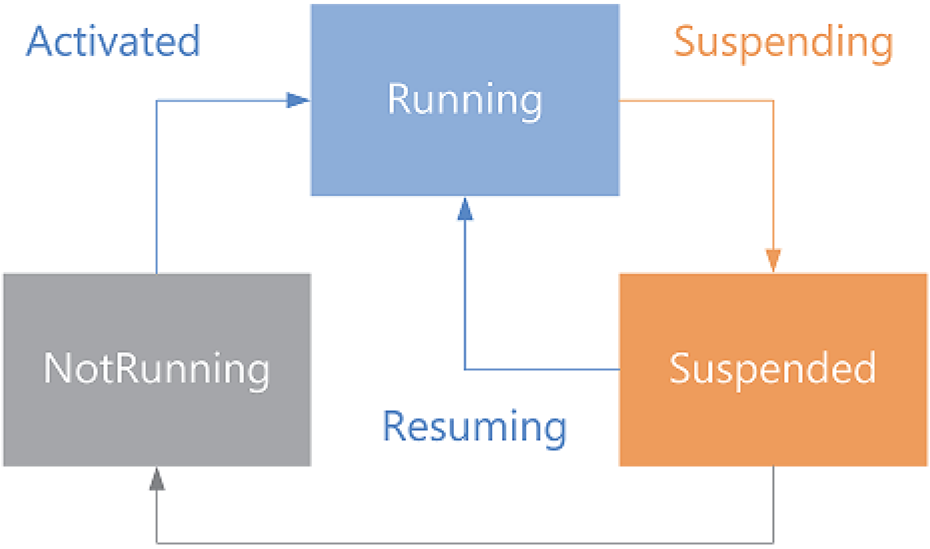


Figure 6: App Lifecycle state diagram[10]

Despite being suspended or terminated these apps are able to register several types of background processing, if proper contracts are in place.

Background tasks allow apps to remain responsive to major events. Triggers include network coverage change, incoming data through connections, system events, push notifications and time triggers. These tasks are run in a separate process from the app for isolation purposes. They are activated by a contract and entry point located in the Windows Registry, as explained before. Other types of background activities which can be registered include transfers of data and audio playback.

Because of the low resource consumption design of Windows Runtime, there are many restrictions as to how often and how frequent these background tasks can be executed. Every app is able to run a time-triggered background process every 15 minutes, and can on a total only consume about 1 second of CPU usage across 2 hours [13].

[10] Source: http://msdn.microsoft.com/en-us/library/windows/apps/hh464925.aspx

2.2.6 App Sandboxing

For security reasons as well as resource control, Metro Style Apps are executed inside a sandboxed environment. The consequence of this is that apps are restricted from consuming system resources without being explicitly allowed access.

The Windows Integrity mechanism was first introduced in Windows Vista, and its main purpose is to restrict access permissions of applications that are less trusted. Regular applications are commonly executed in the medium integrity level while apps are run in a new integrity level called App Container [14]. This container ensures that no app access or modify the system or any other apps directly. By default apps can only access local isolated storage for that particular app.

Most system calls from an app go straight to the kernel while other, more restricted, go through a broker process. This process is responsible for managing the apps system access, in accordance to their specifications in the app manifest, and accesses privileged data and devices on behalf of the app. This mechanism, together with the Windows Runtime Object model, completely isolates apps from each other and the system by preventing less secure code from modifying objects at a higher integrity level.

2.2.7 Windows Notification Service

The Windows Notification Service is Microsoft’s push based notification platform for conveying notifications to Windows 8 enabled smart devices. It enables third-party developers to register with the service and distribute notification from its own cloud services to apps residing in the client machines. Push notification systems are known to be extremely scalable, with little overhead at the client side [15]. Because it is push based, apps do not require polling services to receive updates.

Apps are able to receive push notifications and update their live tile while being suspended, but in reality they do not perform the updates themselves. The OS kernel receives the updates and renders the tiles according to an XML specification of the tile layout without the app interacting. In the event of a resume, the app notices this update and can then request the new information from its third-party service.

Setting up the push notification capability for an app in a third-party service involves the following steps. First, a developer license needs to be acquired, which provides credentials that the developer can use to authenticate his own cloud service with WNS. These credentials consist of a package security identifier and a secret key. Each app has its own set of security identifiers, which guards against sending notifications to other apps. This authentication mechanism is implemented using the Oauth 2.0 protocol[11], and by providing these credentials to WNS, the cloud service is given an access token which it can later use to send notifications.

A client app expresses its interest in receiving notifications by sending a request to the notification platform residing inside the client operating system. This platform then contacts the Windows notification service to create a notification channel. The identifier, in form of an URI, is returned to the caller via the notification platform. The client app then forwards this URI to the apps own cloud service. As illustrated in Figure 7 , when the cloud service has an update to send, it notifies the WNS using this URI which then routes the notification to the client.

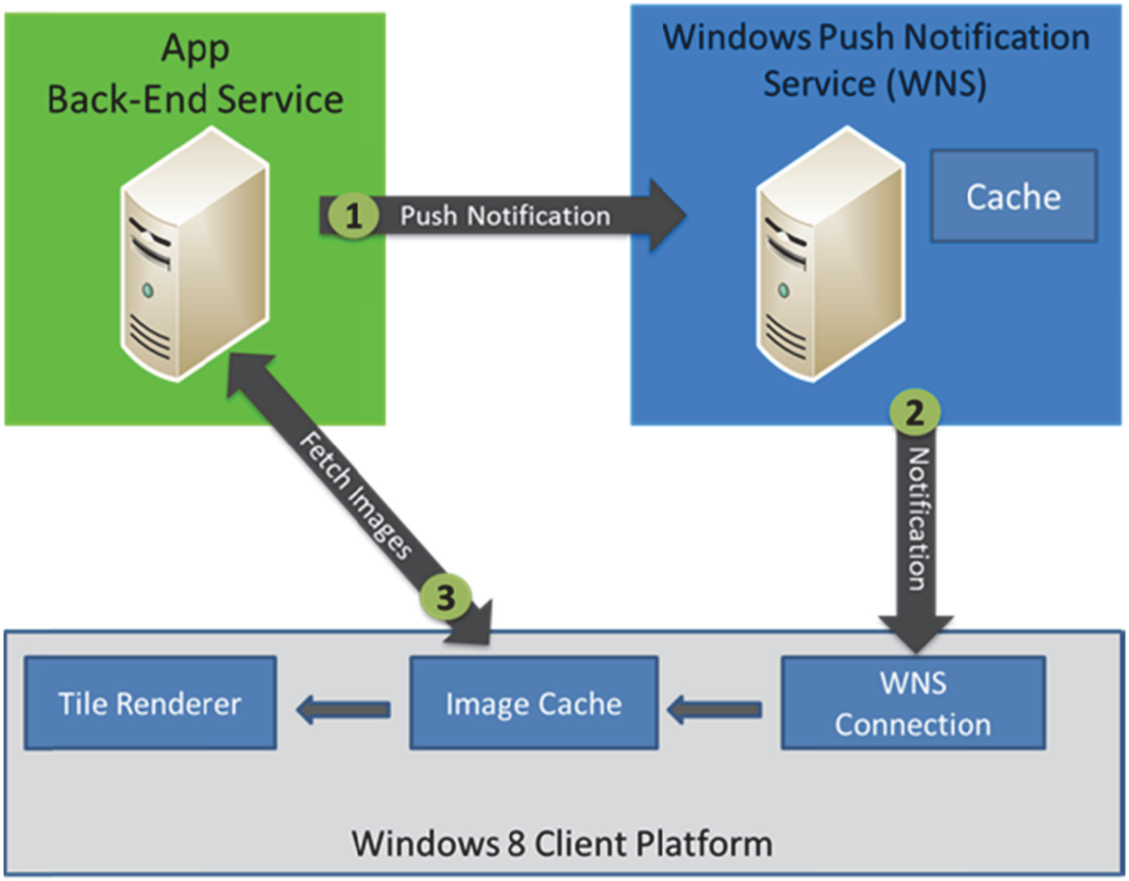


Figure 7: Windows Push Notification Service [15]

2.2.8 Programming model

Windows's emphasis on the design aspects of apps has influenced the programming model as well. Microsoft has stated that all API-calls not serviceable in fewer than 50ms should be made asynchronous. Therefore, Windows Runtime only exposes asynchronous versions of all system services that satisfy this property. One of the justifications for forcing this on developers is to keep the Graphical User Interface (GUI) responsive in the face of high latency I/O. Furthermore,

[11] http://oauth.net/2/

asynchronous programming is known to have its performance advantages, especially with regard to scalability.

The way that the C# projection implements this behavior is by the “await” and “async” operators. All functions which are asynchronous are marked with the asynchronous property. When a function call is marked with the prefix “await”, the function call immediately return to the calling thread. The rest of the code following the asynchronous code is left unexecuted. A callback is then marshaled to the continued execution of the function when the asynchronous method finishes. The function thus executes in a perceived sequential fashion. The returning thread is then able to do other computations in the meanwhile, such as keeping the GUI responsive. This type of asynchrony embedded into the programming language leaves a lot of the dirty mechanics away from the developer. Things like callback handlers and consistency are handled by the runtime and compiler instead.

2.3 Evaluation of platform

This section evaluates the choice of Windows 8 as our example asymmetric system and how it conforms to our requirements. The first part will defend the choice of platform and identify which properties are advantageous to our conjecture. We will then identify and explain the specific parts of the system which restricts our own design, as well as others.

We chose this platform because of its massive distribution and influence potential. Windows 8 is a new version of the most widespread operating system in existence, reaching out to a nearly a half billion users worldwide. Based on our assumptions, the trend suggests that Microsoft is moving towards consolidating all their operating systems and services to provide a unified app platform for all form factors. Therefore using this system for evaluation will give a broad foundation for our conjecture.

Microsoft introduces a new design paradigm with the Windows Metro Style Apps. Devices that have reduced display real estate available will benefit from having tasks more efficiently solved through this. It is our experience that this new immersive no-nonsense design will prove highly usable in combination with our objective of reducing the time used in app installment.

Our requirements also benefit from Windows’s strict resource management. The app lifecycle and the brokered system access guards against excessive resource consumption. For example, having 10 automatically configured apps simultaneously working together without such restrictions would hold a significant overhead, especially on low powered devices. The extended cloud integration, providing new features such as app, data and configuration roaming is very applicable as well.

Although there are some positive traits with using Windows 8, there are also some restrictive properties associated with it that must be addressed. The same resource optimal environment that is inherently a positive trait is also restrictive to our requirements.

The most prominent of these is our requirement for apps to ***communicate***, both internally and across hosts. The only way these apps are allowed to run is in full screen and when not, they are suspended. Therefore, we cannot assume that apps are alive for receiving communication. The operating system do support some type of perceived multitasking by allowing apps to register background tasks, but these are subject to the same isolation and resource restrictions as well. These tasks are able to trigger on important system events and it is our opinion that apps should also remain active in response to events occurring when other apps inside the system change as well.

Apps should be able to cooperate on best aiding the user in achieving his goal, much like the intent system exposed in the Android platform. The contracts available in Windows Runtime aim at serving a similar purpose to this. The major drawback of this is that they must be activated explicitly by user interaction. This invalidates the dynamic properties we seek in our system. A simple example highlighting this inefficiency follows.

Consider a user involved in a work session solving problems at different domains, aided by different app instances. It would be reasonable to assume that these apps should respond to state changes in the others. In Windows 8 Metro Style Apps, this functionality is explicitly prohibited. The user must invoke these commands in different apps in order to keep them consistently operating. This would impact the efficiency and speed, at which the user is able to perform his work.

The app sandboxing together with the Windows Runtime Object Model isolates apps from the underlying system. More specifically, metro apps are not allowed to install other apps. This functionality is only available to desktop applications. This complicates our idea of ***dynamic configuration*** of an entire app environment through metro apps. Furthermore, the Windows operating system will not install any metro app not signed by a trusted certificate authority. Under normal circumstances this would be the Windows Store. But, it does not provide a mechanism for automatically configuring an entire app environment.

Software developers have already expressed their concerns about this closed up platform, one of which is Mozilla. Their intent to make a metro enabled browser for this new operating system is impeded by the restrictions that Windows Runtime imposes.

In a document on their web side [16] they evaluate the runtime and expose the major properties of the model that restricts this, one of which is the lack of Inter-Process Communication (IPC) support. Furthermore, there is no mechanism for dynamic execution of code, making it difficult to run a scripting engine inside the browser. Mozilla has evaluated the built in browser, Internet Explorer 10 and discovered that a lot of the restrictions put into Metro Style Apps are effectively avoided by this app. This on the other hand proves that there are use cases and workaround for a lot of the limitations in existence. They therefore restrict third-party browsers from using the same capabilities as their solution does.

2.4 Summary

In this chapter we have detailed the app platform type as a type of asymmetric system. We have identified the most important components in this platform and how they interact. We have further evaluated the major components of Windows 8, the subject app platform we base our conjecture upon.

Two major focuses in the design of this new app platform has been low resource consumption and security. Because of this, Windows 8 poses several hindrances to our initial conjecture. Automatic configuration of apps onto a client system is prohibited in this platform. Apps run inside a sandboxed environment where they are isolated from the system and other apps. Access to resources is prohibited unless explicitly allowed and, because of this, IPC mechanisms are not available for apps. This is also further hindered by the fact that apps are suspended by the system when not running in full screen. This means that usually only a single app is allowed to run at a time.

3 System Architecture

To overcome the limitations of Windows 8 and evaluate our conjecture, we have designed Javza[12].

This system enables dynamic configuration of app environments based on simplified contextual data retrieved from the user. Moreover, it supports the integration of apps residing on the same host as well as across hosts by means of communication. This chapter details the architecture of the system, explaining the function of the different components and how these interact.

The overall architecture of Javza is based on a three-tiered model, consisting of the client side, server side and storage server. The three components are depicted in Figure 8. At the client side we have the Arbitrator which manages all app instances and provides integrating functionality for these. At the server side we have two components, the App Provider and the Group Recommendation Service (GRS). Both these use our storage server for managing structured data such as user context and app packages.

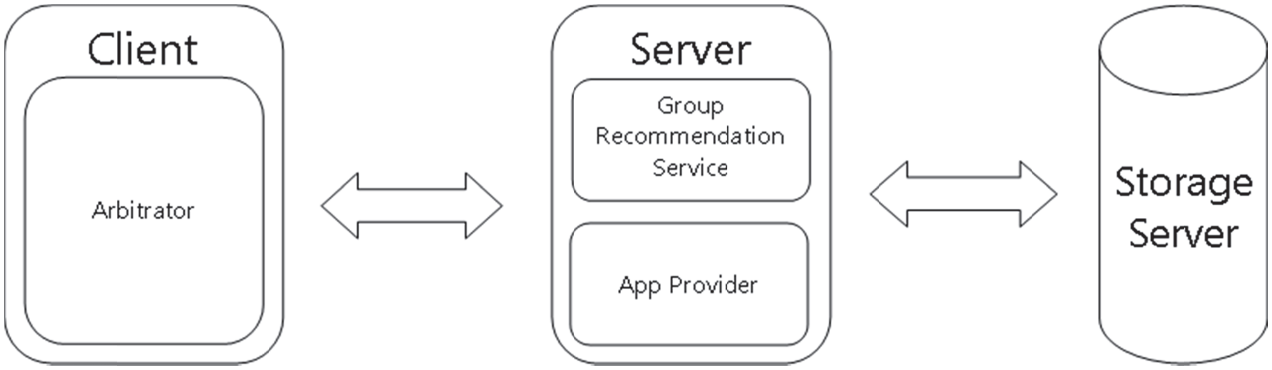


Figure 8: Architectural Overview

3.1 Server Components

The server side of Javza includes two components. The App Provider, which conveys apps to clients automatically based on contextual data received. The GRS, which provides recommendations for group assignments based on contextual data.

3.1.1 App provider

One of our motivations for designing Javza is to enable dynamic app configuration. By this we mean the ability to automatically download and install apps without the explicit involvement

[12] The name comes from the language of the indigenous Sami-population located in the northern-most part of Europe. The meaning behind it is a special type of singer, referring to a virtuous who masters the most challenging song techniques.

with the user. To be able to do this, we require a server responsible for managing and distribution of the app packages. For this purpose we introduce the App Provider.

As mentioned in 2.2.3 app stores commonly require users to explicitly download apps from the store as is the case with the Windows Store. The conceptual model of this is similar to the one depicted in Figure 1.

The interaction with these goes through several steps. Users perform a search through the app store and upon finding what types of apps are necessary, they download the app. The app is then validated by the user and installed on their local client. This process is time consuming and perhaps not very efficient, especially on restricted display spaces quite common for smart devices. This requires us to rethink the model and introduce an alternative, more efficient approach.

We have designed a service resembling that of the Windows Store. However, the Windows Store also requires the user to explicitly search for, download, validate and install apps. To adhere to our initial requirements we instead implement a model which installs apps automatically, pushing the apps to the end users instead of pulling them. To differentiate between the two types we refer to our model as an App Provider.

Automatically downloading and installing apps do not come without its consequences with regards to the integrity and security of the client system. Downloaded executable code is generally considered very unsecure and Windows 8 enforces strict policies to guard against this from occurring. Furthermore, this approach to app management could provide issues with privacy and could be considered intrusive if not properly handled. We abstract away these issues here as it is outside the scope of this thesis. However, we acknowledge the need for secure mechanisms for app management, and reason about the issue later in section 7.1.

The notion of pushing apps to clients requires us to have a fair understanding of what type of apps the user needs. More specifically, we are required to, with high probability and precision infer what apps the user needs. Accurately predicting user needs is difficult and we will not go into particular detail on this, as it is beyond the scope of this thesis.

We instead adopt a rudimentary description of context to infer user needs. Contextual data can be anything that describes the environment and situation the user is involved in. This data could be explicitly provided, such as the case of the common search engine query. Or it could be implicitly inferred by available context. Mobile devices contain an abundance of sensors, able to provide implicit contextual data. These include GPS link, Altimeter, Magnetometer, Accelerometer, Microphone, Video Capture and Image Capture. Our solution is confined to explicit context, retrieved through the specification of intent, embodied in Javza as strings.

Based on this contextual information provided by the client, the App Provider performs an analysis and matches it to app packages residing on the storage server. These are then returned to the client and automatically installed. The contextual information is stored in a per user fashion in the storage server for further use by the GRS. The overall interaction between these three components can be seen in Figure 9.

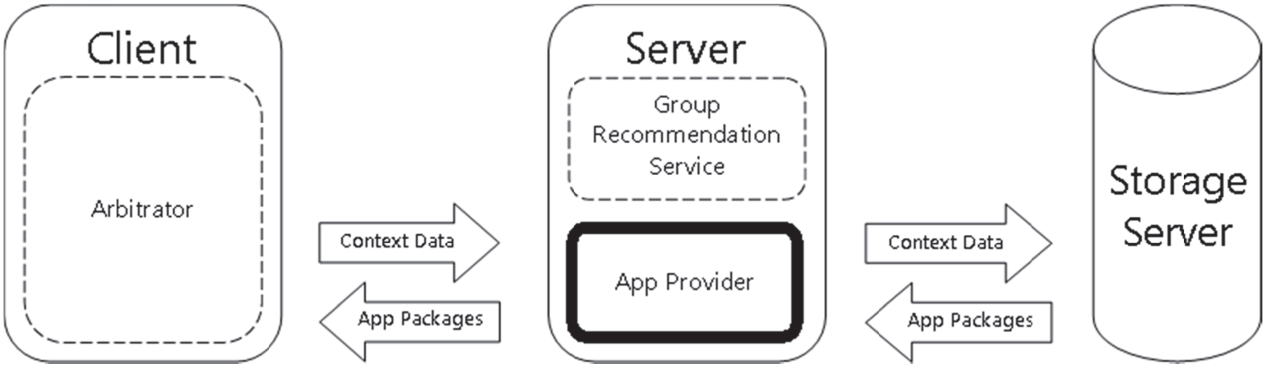


Figure 9: App Provider interaction

3.1.2 Group Recommendation Service

We want to provide the ability for apps across different hosts to communicate with each other. Apps used inside the same domain should be able to share data to aid one another in their purpose, despite host boundaries.

To be able to provide communication between apps across different hosts, we need a way to identify which hosts should be able to communicate. The objects participating in the integration needs to share some similar intent to validate this integration, we therefore need a mechanism to detect this. To provide this service in Javza, we introduce the GRS.

These recommendations could be made implicitly or explicitly. In the explicit scenario, users are able to decide what users they are connected to. The other recommendation type issues implicit recommendations which automatically connect users.

To limit the scope of this thesis, we abstract away the notion of users consent to participate in a given group. Our purpose is not to evaluate the security and privacy aspects of such a system but rather provide the ability for apps to communicate across different hosts. We have not deigned an optioning to allow users to manage group access. Hence we declare that every active user of a host system is willing to connect to groups of other users. And we therefore connect users automatically without user involvement.

The GRS's sole purpose is to provide users with recommendations about groups of other users with similar intents. It retrieves the contextual information from the storage server and performs an analysis on this information. If the analysis concludes that there are groups of users with similar intents, a recommendation is sent to these users. These users are now by the information contained in the recommendation, able to connect into a group. Figure 10 illustrates the push based information flow and placement of this component.

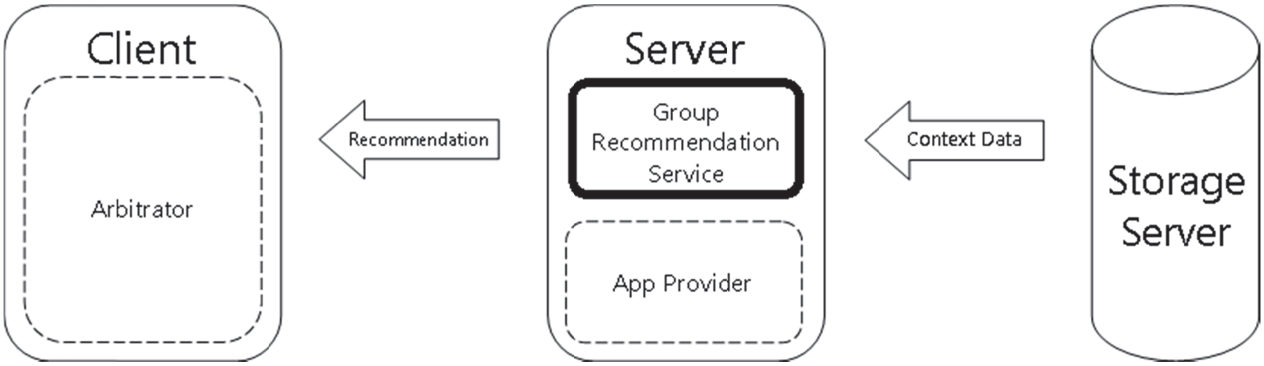


Figure 10: Group Recommendation Service

3.2 The Arbitrator

By allowing apps to share data, both within the same host and across hosts, they become providers of content for other apps. Apps would not only rely on user input to that specific app, but also user input and data generated in the context of other apps. This can provide insight from their specific domains not otherwise available.

Consider two example apps in a single host system. One of them is a restaurant listing app with functionality for reserving tables. The other is a map app with GPS capability. The user interacts with the restaurant app, first searching and then booking a table at a given restaurant. The user then checks the map app to find the restaurant. In the case where these apps aren't able to communicate, the user must then perform the same search in both apps. This causes unnecessary complication and time consumption, especially when the host system has a restricted display surface such as the case with smart devices. Furthermore, consider the same scenario across two hosts. Two particular users are performing a similar task. They are cooperating on finding the solution to the given problem. The apps on each users host should also be able to communicate for the benefit of this collaboration.

It is our conjecture that it will prove beneficial to the overall functionality of the system and provide a richer user experience. As mentioned in subsection 2.1.1 the Android platform provides this functionality for apps within the same host. In subsection 2.2.6 we explained how Windows 8 restricts us from doing this implicitly.

To satisfy our requirements and evade Windows 8's app sandboxing restrictions, we have designed the Arbitrator. This naming comes from its function as a decision maker and mediator between apps in Javza. This runtime is located within each client host and consists of multiple cooperating components.

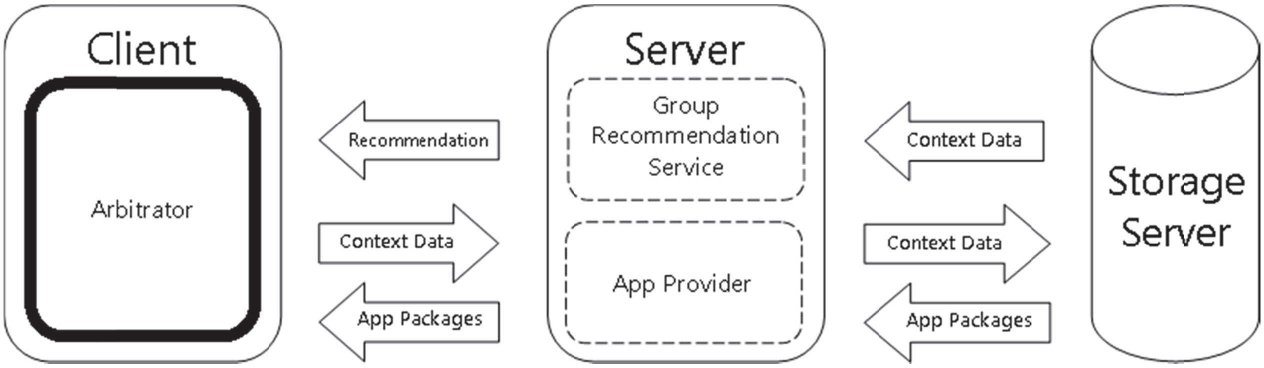


Figure 11: Arbitrator placement in architecture

It provides a mediation service by Inter-Process Communication between apps on the same host, and networked communication service for apps on different ones. Figure 11 illustrates how a single Arbitrator instances, located on the client, interacts with the rest of the system. It forwards contextual data back to the App Provider. When it receives App Packages from the App Provider, it automatically installs these on the client system. When a recommendation is issued to it by the GRS, it then connects to a group of other Arbitrator instances.

3.3 Storage server

Both the App Provider and the GRS requires a storage server which provides consistent and structured storage of data, accessible by both. The GRS needs this data to keep track of the groups in effect and users associated with these. The App Provider will store apps and manage the contextual information provided by each user for analysis.

The storage server also maintains a correspondence between the groups, and what apps are installed by which user of a group. The App Provider will need to control which apps each user has installed already to avoid duplicate access to apps. Each app is also associated with a specific context, specified by the developers of them. Apps are collected together into sets of apps, which corresponds to similar usage comparable with contextual information. As mentioned previously, the notion we adopt in this thesis is very rudimentary, for abstraction purposes. In Javza we use simple textual strings to represent a user’s contextual information.

3.4 Overview

All these components interact to provide the joint functionality we require in Javza. Figure 12 describes the architecture in use and the interaction between the components. This interaction consisting of the following steps:

1. The Arbitrator runtime provides contextual data to the App Provider.
2. The contextual data is evaluated and stored in the storage server.
3. The App provider then retrieves the collection of apps that best match this from the storage server.

4. These apps are then sent back to the Arbitrator residing at the client, who then automatically installs these apps into the system.
5. Periodically, the GRS retrieves the contextual information from the storage server and analyzes it.
6. If it discovers clients with similar contextual data, it sends a recommendation to these users to join up in a group.
7. Once inside this group, the apps on the host systems of these users are now able to communicate with each other through networked communication.

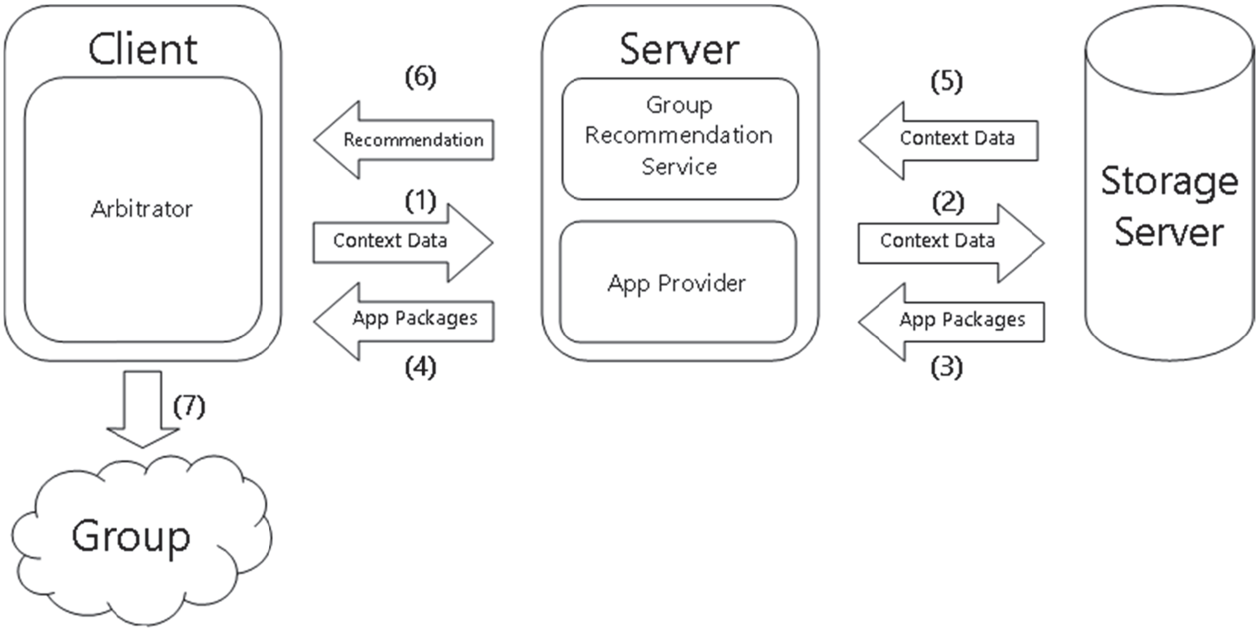


Figure 12: Architecture in use case

3.5 Summary

In this chapter we have identified the separate components involved in the design of Javza. We have explained our design and how it relates to our conjecture and the restrictions imposed by Windows 8.

We have introduced the concept of an App Provider. Consumers spend a lot of time in traditional App Stores by searching, downloading, validating, installing and then configuring apps before they are ready to be used. The App provider aims to remedy this by installing apps automatically based on simplified contextual information retrieved from the user. There are some obvious security implications with installing apps automatically onto a system from across networked connections. In designing Javza, we abstract away these issues and rather focus on how to provide this functionality.

We explained the benefits of allowing apps to share data, and contribute to a richer user experience. The Arbitrator has been designed to complement Windows 8, and provide this feature both within and across hosts.

To allow related app instances across users to be able to share data, we have designed the GRS. This service issues recommendations to users with similar intents to join into groups. The service enables users of users to communicate implicitly.

We have identified the most relevant requirements for the storage server in order to provide this functionality. Finally, we provided an overview of the different components interacting with each other to fulfill our requirements.

4 Implementation

This chapter details the prototype implementation of Javza, which we will further use to evaluate our initial conjecture. In section 4.1 we detail the server side components which provide group recommendations and apps to clients. Section 4.2 then explain the Arbitrator implementation, our client side runtime which provides automatic configuration and integration of apps.

We should mention the difficulty of researching a yet to be released proprietary system such as Windows 8. In parallel to our research efforts, this system has been released in multiple versions. To adapt we have made modifications to our system during this phase which will be detailed later in this chapter.

Windows Runtime and Windows 8 gave us four language choices for implementing our prototype, C++, Visual Basic, JavaScript and C#. Visual Basic was deemed too old a technology and JavaScript too restrictive. C# was then chosen based on a tradeoff between performance and speed of development. Although native code execution is faster than managed code execution, prototyping in C# is much faster than the native alternative C++. The underlying frameworks used are the .NET platform and the Windows Runtime.

The custom communication protocol we implemented uses the same underlying messaging format throughout the entire prototype implementation. It is implemented exclusively on top of regular TCP which gives us total control of the performance. We exclude unwanted functionality by stripping the communication down to the bare bones. Other communication mechanisms such as web services and Remote Procedure Calls are more costly if not used efficiently and without expert knowledge. We are quite familiar with TCP sockets and they provide the basic properties we require in communication channels, reliability, synchrony, and congestion control.

To assure interoperability across different types of Metro style Apps, we should be able to share structured data. To be able to share structured data across stream based communication primitives such as TCP we must serialize data down to a byte representation. Metro Apps and .NET are able to serialize objects by two different approaches JSON and XML. We use XML as the method of serialization in our implementation.

4.1 Server Side

We will now explain the implementation of the server side of Javza. Although the design indicates that the App Provider and GRS are separate components, both run inside a single process in separate threads and use the same storage server.

4.1.1 App Provider

Our motivation for designing the App Provider is to enable dynamic app configuration. We redefine the app store model by installing apps automatically to the end user. In order to accurately do this we should infer the needs of the users based on contextual data and associate these with apps.

This component manages app packages and associated tags describing the apps usage. When submitting an app to the App Provider, the developer is responsible for including these tags in form of strings. Apps are then based on these tags grouped into sets. These sets will then contain apps with similar capabilities.

Implementing an interface for app submission is outside of the scope of this assignment. The current implementation is statically configured with apps that are tagged and stored into the database. Furthermore, our focus for this thesis is not on app development so to limit the scope we refrain from developing many complex apps. We have however implemented 5 apps which use Javza, with related functionality. These apps are based on a developer sample retrieved from the Windows Development Center[13], although extensively altered. We describe their purpose later in chapter 6.

Since we have only implemented these apps for evaluating Javza, there are some restrictions to our initial design. When a client provides contextual information to the App Provider, all 5 apps are returned for installation instead of particular set of apps corresponding to this. However, the App Provider supports grouping apps into sets using the mechanism detailed below in subsection 4.1.4, although our implementation does not utilize it.

Furthermore, the App Provider also collects usage information provided by the clients of the system and stores them together with the apps located in the storage server. These statistics come in three types on a per-app granularity: the installation count, an uninstallation count and the number of activations. This information could help precision in searching for apps by providing information valuable in ranking. However, because of the same reason mentioned above we refrain from using this functionality.

When a client reports back some contextual information, the App Provider then stores this and returns the apps to the client for installment. The package format of these apps is identical to the ones downloaded from the Windows Store, as explained in subsection 2.2.3, except that these are signed by our own temporary certificate. Upon compilation of the apps, the Visual Studio IDE creates a temporary certificate for the app, and signs the app package with it automatically. It also creates a batch script for installing both the app and the certificate into the client system. The app packages themselves are actually stored on disk, while the metadata

[13] http://code.msdn.microsoft.com/windowsapps/

resides in the storage server. To convey multiple app binaries across the network we compress them together using the DotNetZip[14] library. This zip archive file is sent back to the client, which in turn decompresses and installs the apps.

4.1.2 Storage Server

Our requirements dictate that the storage server should hold structured data. As depicted in Figure 13, the data requirements we presented in section 3.3 map fairly well to the entity-relationship model. The schema of this component is a relational database schema implemented through the ADO .NET Entity Framework[15] on top of a Microsoft 2012 RTM SQL Express Server.

A group entity contains more user entities which in turn have app entities related to them. Associated with each app entity is a set of tag entities which describe the usage of that app. The relationships between the tag and user entity shows that we use the same entity for storing the contextual data retrieved from users. App entities are also related to their own groups, which the schema refers to as a taxonomy entity.

By using the entity framework, we are able to manipulate relational data in an object oriented fashion. Furthermore, because of its integration with .NET we are provided with LINQ query language support for retrieval of data. We express these queries directly in C# code, and the underlying system handles the execution transparently.

[14] http://dotnetzip.codeplex.com/
[15] http://msdn.microsoft.com/en-us/data/ef.aspx

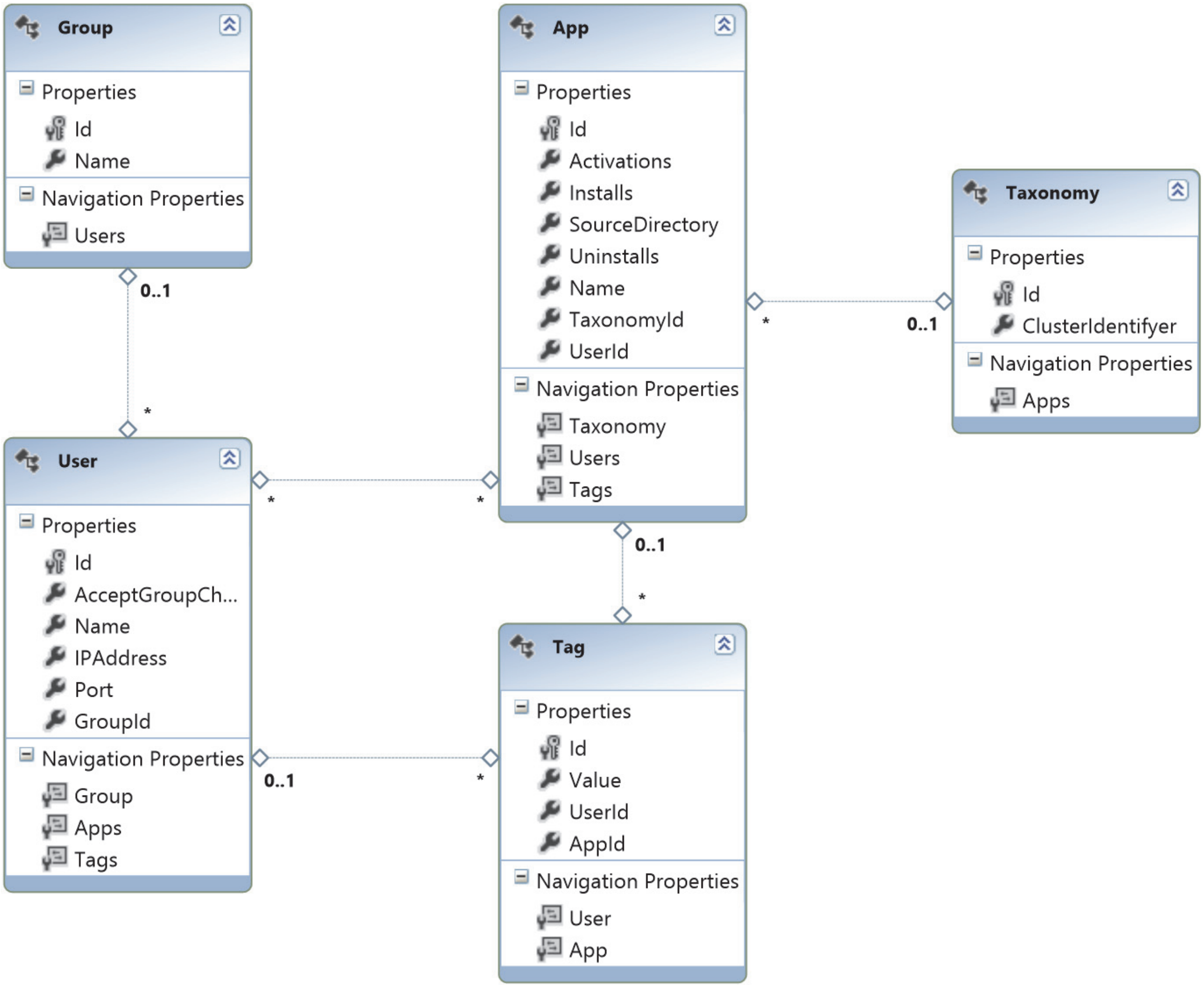


Figure 13: Data Base Schema

4.1.3 Group Recommendation Service

Our motivation for implementing the GRS is to aid integration of apps across hosts. The GRS contributes by recommending users to groups which then implicitly enables apps within these groups to communicate with other apps. To realize this we need a method of determining what groups users are recommended to join. More specifically, to determine what users have similar intents from the contextual information gathered by the App Provider.

The GRS performs analysis on this data retrieved from the storage server within regular intervals. The data is consumed by an analysis algorithm which will detect if users with similar intents exist. The recommendation is then pushed out to the users, along with enough information to connect the group. The storage server is responsible for storing this contextual information for each user of the system, and also the associated groups they currently belong to.

As explained in 3.1.2, GRS does not reason about users consent to participate in a group. In a real life scenario we would have to address these privacy implications but as it is outside the scope of our thesis, we abstract away from this issue. In our system, all active clients are interested in being connected to a group.

User records are admitted into storage on a first encounter policy. The first time a user contacts the App Provider he is admitted into the data records and is then able to receive recommendations from the GRS.

4.1.4 Detecting groups

This subsection describes the algorithm used by the GRS for detecting objects with similar contexts. Our choice of method for analyzing contextual data comes from two basic requirements. The method should provide a more sophisticated analysis than simply comparing values in a binary form. Furthermore, it would be considered a benefit if the approach is relatively straightforward to implement.

The same approach is used in the App Provider to group apps together, but as explained before we do not utilize this. In the following subsection we refer to them both as objects.

We adopt a rudimentary approach to contextual information where the information gathered from these objects consists of distinct tags, more precisely string values. The algorithm employs techniques that are inspired the Vector Space Model [17] for computing document similarities of these tags.

To assign objects to groups corresponding to their contextual information, we look at each object’s recorded tags as a vector in Euclidian space. We then use a learning algorithm to find similarities, and use this to group together objects based on data clustering.

Each unique tag corresponds to a single dimension, and is encoded in the object’s single vector as either one to mark the presence of a term, or zero for the absence of this term. The amount of dimensions corresponds to the entire amount of unique tags across all evaluated objects.

In other cases where contextual information is actually quantifiable this approach is not necessary. They are then directly adaptable into the vector. An example of this could be the latitude and longitude coordinates retrieved through the GPS sensor on a smart device.

The algorithm used for determining correlation between the vectors is the k-means algorithm [18]. It is a classical clustering algorithm which classifies output cluster patterns from input samples.

It starts out using k initially chosen vectors in the Euclidian space. These are chosen either from the set of input vectors or randomly distributed throughout the space. For each vector evaluated the algorithm computes the Euclidian distance from that vector to the k vectors in the cluster. It then assigns the vector to the closest one, and re-computes the center of that cluster by arithmetic mean. The process continues until the clusters converge towards a single point. In our case where the amount of input vectors is finite, this process continues until all vectors have been evaluated.

For an n-dimensional space, the Euclidian distance is given by the equation:

$$d(\vec{p},\vec{q}) = \sqrt{\sum_{i=1}^{n}(\overrightarrow{q_\imath} - \overrightarrow{p_\imath})^2}$$

And the arithmetic mean v for each dimension, for each vector $\vec{u}$ assigned to a cluster is given by:

$$v = \frac{1}{m}\sum_{x=1}^{m}\overrightarrow{u_x}$$

The key to this approach and the most difficult part is choosing how many initial clusters exist in the data set. We have no way of knowing the amount of natural clusters in advance. "*Increasing the number of clusters will always decrease the amount of error in the result, converging in the case where each cluster contains only a single point. Choosing k should strike a balance between maximum data compression in assigning the data to a cluster, and the accuracy corresponding to the original value of the point*"[16].

For simplicity we implement the following approach, commonly used in text database analysis [19]. Here all the input vectors to the k-means algorithm are put together in a matrix D where each row corresponding to a single vector. The number of clusters is determined by the formula: $k = \frac{m*n}{t}$ where t is the number of non-zero entries in D, m is the number of vector points and n is the number of dimensions.

[16] Paraphrasing: http://en.wikipedia.org/wiki/Determining_the_number_of_clusters_in_a_data_set

We can by this approach extract the vector assignments and group users accordingly. Consider the dimensions $D = \{cheese, wine, alligators\}$. We are then operating in 3-dimensional Euclidian space.

We have three objects with corresponding contextual information:

$$U_1 = \{cheese, wine\}$$
$$U_2 = \{wine, alligators\}$$
$$U_3 = \{alligators\}$$

Their corresponding vectors will then be:

$$\vec{u}_1 = [1,1,0]$$
$$\vec{u}_2 = [0,1,1]$$
$$\vec{u}_3 = [0,0,1]$$

We then determine k as $k = ceil\left(\frac{3*3}{5}\right) = 2$

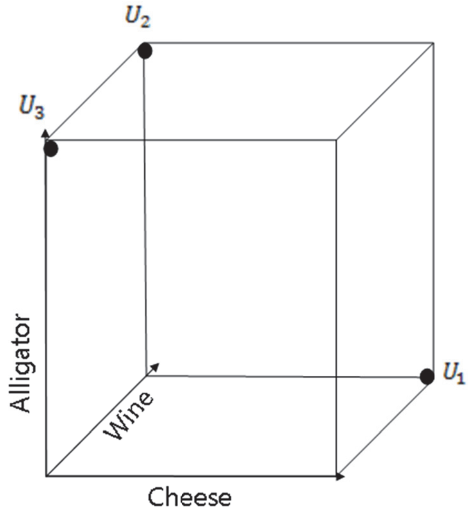


Figure 14: 3-Dimensional terms

We observe from Figure 14 that the closest terms are in fact $U_2 = \{wine, alligators\}$ and $U_3 = \{alligators\}$. These two are only separated by a single dimension. $U_1$is diagonal across the x-y plane from U_2 and diagonal across all three dimensions from U_3.

In our example we have only three dimensions and three input vectors for illustrative purposes. However, the method is applicable for any number of dimensions.

Normally in vector space models the cluster center point is not computed as an arithmetic mean of the vector coordinates, rather they use angles between vectors. Furthermore measuring the Euclidian distance as a method of comparison is crude. Since it is out of the scope of this thesis to evaluate sophisticated document analysis methods, we refrain from implementing this.

The approach taken here boils down to a comparison of values represented in vector form. A more sophisticated alternative to detecting objects with similar interests would be to use semantic textual analysis. In such an approach, one would instead try to infer the meaning behind the tags. One could for example use a lexical database such as WordNet[17]. WordNet records the semantic relations between different words in the English language into a database. By this approach one could discover how far objects are relative to the meaning of the terms associated with these objects. This relationship can be quantified and used to compare similar intents.

4.2 Arbitrator

This section describes the implementation of the Arbitrator, the client side runtime of Javza. Our design dictates that we require a runtime which can automatically install apps. Furthermore, it must function as an integration point between apps at the same host and those receding at different hosts. In order to fulfill this purpose it must remain in continuous operation regardless of apps being suspended by Windows. The Arbitrator is therefore implemented within in the desktop environment and not the Windows Runtime environment.

It consists of an app management component and a communications component. Though they fulfill different requirements of the system, their separation is at a logical level, meaning they are both located inside a single process.

4.2.1 App Management

The App Management Component (AMC) is in charge of installing and monitoring apps received from the App Provider. Administering communication between apps requires proper addressing, and in order to address apps we need to know which ones are installed. If a user should choose to uninstall an app our runtime should reflect these changes. Since the App Provider pushes apps to the clients it should be notified of this event, to record what apps are already associated with users. Furthermore, knowing how each app is used in our system could prove an important contribution when ranking these. The AMC therefore consists of two mechanisms, one that provides dynamic installation of apps and one that performs monitoring.

To manage app packages we use the Windows Package Manager. This is an operating system provided asynchronous API which enables us to query, install and remove Metro style App packages from the system. This API is only available for desktop applications, which further validate our decision of implementing the Arbitrator outside the Windows Runtime environment.

[17] http://wordnet.princeton.edu/

In subsection 2.2.3 we explained the difficulties of installing apps outside the Windows Store and that packages must be signed by a trusted certificate. However, Windows provides two mechanisms that allow developers to deploy their temporary applications while in development on the local system,

For this purpose, the Visual Studio IDE creates temporary certificates for app deployment on local computers outside the Windows Store. It signs these packages with this certificate and upon deployment the certificate is installed into Windows Trusted Root Store. The compilation of app packages include a batch file script for installing both the certificate and the app at the same time, and the AMC provides functionality to execute these with each app package. However, our requirements imply that the installation process should be as swift as possible and installing certificates into the Trusted Root Store costs time. We therefore only installed the certificate in which all our apps packages are signed with, once per operating system instance.

To install these custom apps we also need to have a developer license registered on the system. Windows provide an API which enables us to check for and acquire developer licenses to the system. We automate this procedure and check whether a developer license is already registered upon initialization of the Arbitrator runtime and if not, we acquire one. This must be performed within the context of a developer account, so the system then prompts us for registering with the developer credentials. The procedure is a one-time event per operating system instance, and is only necessary the first time a user instantiates the runtime. With this in place the AMC is now able to install apps issued by the App Provider, as illustrated in Figure 15.

We monitor these apps by querying the Package Manager within regular intervals, retrieving a list of installed apps. We also keep track of the lifecycle of each app installed by monitoring the state of the processes associated with each app. These two approaches give us a fine granular knowledge of when a distinct app is activated, terminated and uninstalled. In the face of such an event, the client runtime reports back this information to the App Provider. The App Provider then stores the information together with the corresponding app.

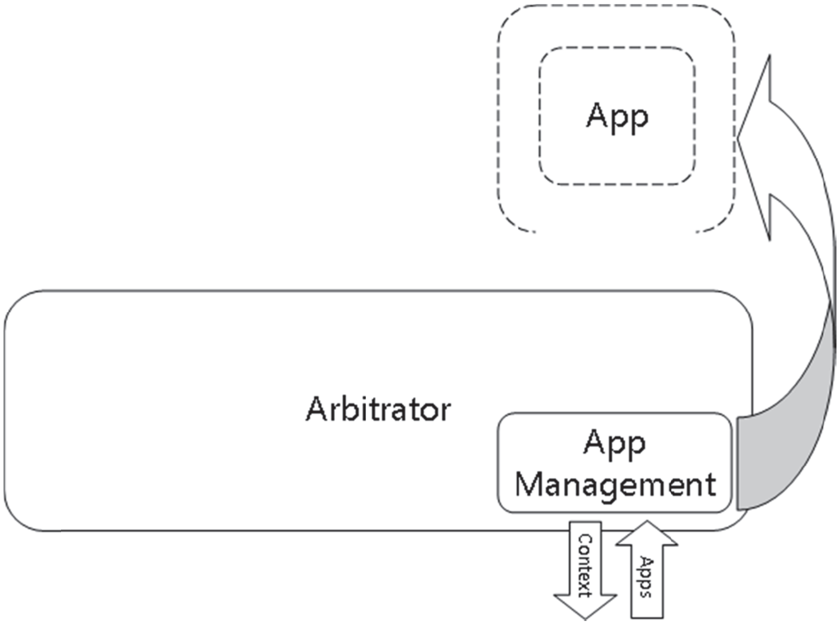


Figure 15: App Management Component

4.2.2 Communication

We conjure that there are benefits in providing integration between apps located on the same host as well as on different hosts. Allowing apps to share data and contribute to the functionality of other apps without explicit user involvement could provide a richer user experience. Furthermore, it would alleviate much of the time consumed in explicit user interaction on restricted display surfaces commonly present in smart devices.

This applies to different smart devices as well, possibly aiding a group of users in a cooperative task. Apps located on the different devices should be able to share data without user involvement to efficiently aid the cooperation.

We here detail the implementation of the Communication component supporting this in the client side of Javza. Figure 16 illustrates the organization of this component residing within the Arbitrator. This subsection is structured as follows:

First we state our requirement with regard to inter-host communication, before detailing our solution to this. We then proceed by identifying the properties we seek in Inter-Process Communication, limitations posed by the underlying system, and how we solved this. Finally we present the naming format used by the Arbitrator runtime to convey messages.

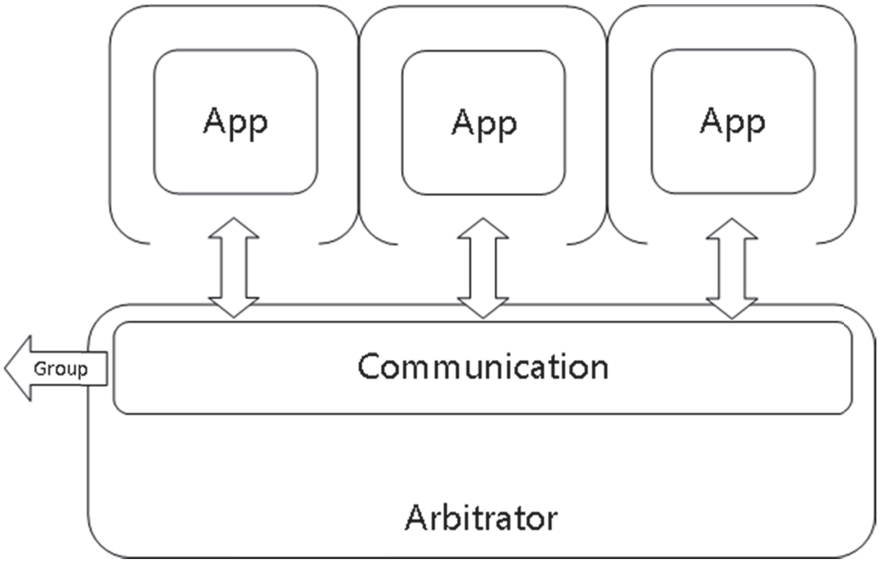


Figure 16: Communication

4.2.2.1 Inter-Host Communication

We conjure that there are use cases where apps located on different hosts could benefit from being able to share data. Furthermore, we anticipate that not all scenarios are efficiently servable from a centralized approach to mediate this communication.

The GRS, described in subsection 4.1.3, provide a service which identifies users with similar intents, and subsequently sends a recommendation for them to join each other in a group to enable communication between apps on different hosts. Figure 17 illustrates the abstract function of this in between Arbitrator instances.

Network connectivity issues are common in mobile environments, and there is the possibility for smart devices to be disconnected from the rest of the network. For the sake of abstraction, we refrain from implementing a rigorous scheme for fault tolerance in this thesis. Even just establishing whether or not a connection is just slow or dead is impossible. We have not formally tested our capabilities of handling failures, and as such we do not support it.

Arbitrator
Network
Arbitrator

Figure 17: Abstract functionality

4.2.2.1.1 A Centralized Approach

In this thesis we have chosen to explore a decentralized approach to inter-host communication between apps. Centralized client-server solutions require a server to relay messages between hosts. We conjure that not all scenarios can be efficiently solved using this model. Examples of this could be a session where the users are within a close range, such as in a collaborative setting. This would be even more pressing if the users are within a network with limited coverage or a metered network. In metered networks the payment plan of a mobile subscription dictates how much data could be transferred, and at what cost.

We consider a case involving the Windows 8 app platform where one app wishes to communicate with another app located on different devices. In Windows 8 apps are suspended when not in use and can therefore not receive communication. However, notifications about important events could be relayed through the Windows Notification Service to these devices regardless of app state. As mentioned in subsection 2.2.7 only third-party cloud services can relay notifications to client systems. So any request for notification to a particular app must go through this. Figure 18 details the communication necessary in order for these two devices to communicate. We list the steps here:

1. First the initiating app (C_2) on the client smart device sends the message to its centralized server or cloud service.
2. This service again, through Windows Notification Service, sends a notification to the other participant (C_1).
3. Upon the receipt of the notification at the clients system (C_1), the system or the user explicitly notices this event.
4. It can then by polling the server retrieve the actual message from the server.

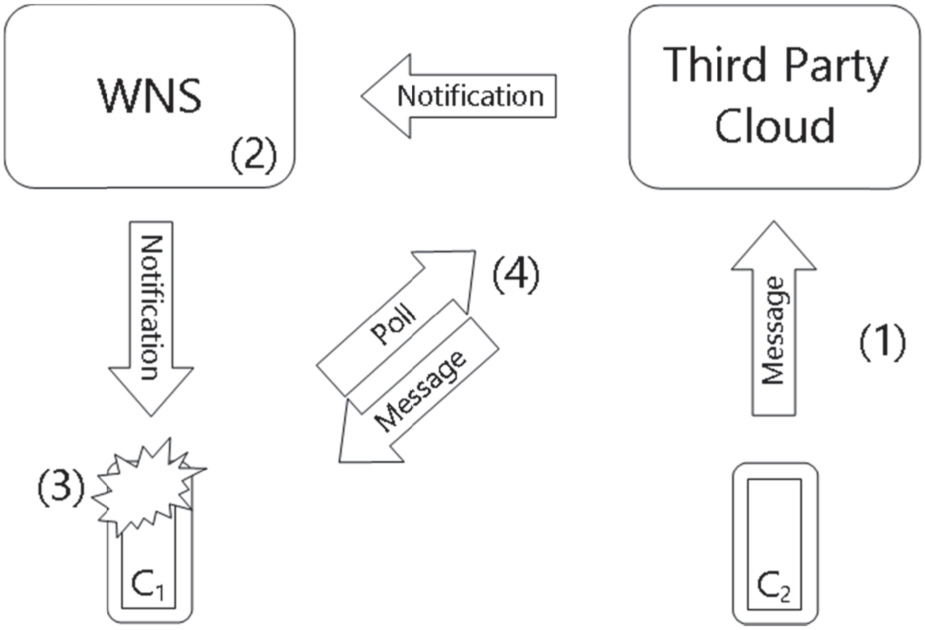


Figure 18: Centralized Approach

4.2.2.1.2 Peer-to-Peer

The alternative we have chosen to a centralized solution is a decentralized approach. Among these the most viable for our use is peer-to-peer communication. The organization of this could be either structured or unstructured. They can also be completely distributed or a hybrid approach with a centralized organizer.

We chose to implement a completely decentralized and highly structured peer-to-peer approach to communication in Javza. The source of inspiration for this peer-to-peer implementation is the Chord peer-to-peer lookup service [20], built for distributed hash tables. More specifically, the prototype we implement resembles the lookup service and organization of nodes in Chord to provide communication between hosts in Javza.

We further build our arguments for the choice of this design upon the similarity with Chord. We cannot argue solely on behalf of our own implementation, because we do not evaluate a lot of the aspects which justifies this choice. We simply note that Chord provides the properties we seek for in communication, and our implementation aims to mirror this.

The first and foremost property is the scalability Chord provides. A vast amount of smart devices are able to be interconnected directly into a peer-to-peer network, without a centralized bottleneck in the system. Furthermore, because this approach is completely decentralized, we are not dependent on any server for communication. The Chord structure, as we will note, maintains redundant links and is therefore able to maintain connectivity in the group despite of nodes failing. This same structure provides relatively good upper bound guarantees as to how many hops are included for a communication to reach its destination. In the event where multiple apps on separate hosts are able to consume a given piece of data shared within the group, the overlay network should support broadcasting. Peer-to-peer approaches provide very efficient ways of broadcasting. This is done by distributing information or state to multiple receivers through epidemic protocols such as gossip based dissemination.

In our implementation, as with Chord, all participants in this network are coordinated into a ring where their placement in the ring is guided by a unique key. The next consecutive key held by a node in the ring is called that participants successor, and the key before called its predecessor.

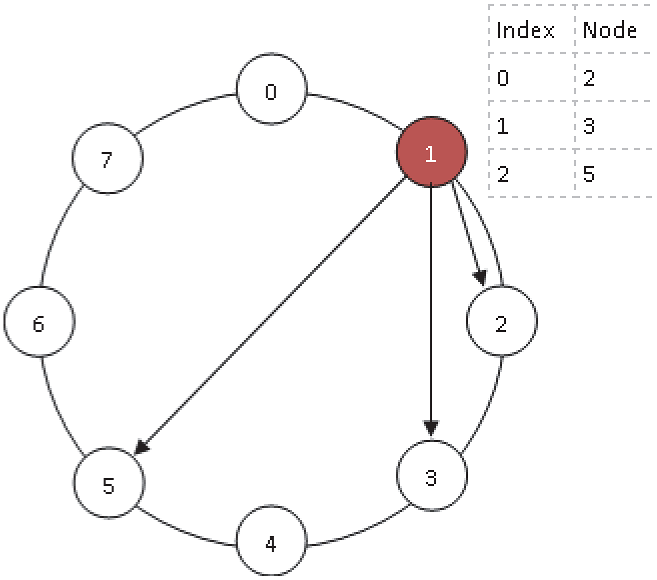
4.2.2.1.3 Lookups

Each participant in the ring holds a table of size m, internally referred to as a finger table. The ring can hold a maximum of N participant where N is exactly 2^m for any given m. This table could be compared to an address book for quick access to other participants in the ring. To obtain the properties explained later in this subsection we use a formula to compute which participants should be present in the finger table. The node address residing in the **i***th* entry of the finger table of node n is calculated by the given formula:

$$x = \left(\left(n + 2^{i-1}\right) mod\ 2^m\right)$$

X is the key of that particular node. It requires exactly 2^m keys for an m length finger table.

Each participant is assigned a set of keys between its own and its predecessor. We refer to this as a key range. When the ring is full, as illustrated in Figure 19, each participant only owns a single key. The number of entries in the finger table is 3, which corresponds to a maximum of $2^3 = 8$ participants



Index	Node
0	2
1	3
2	5

Figure 19: Peer-to-peer organization

The algorithm for looking up a specific key in the ring includes the following steps (Special care must be taken when crossing the 0 key):

1. The user credentials of the recipient hashed down to a given key, and thus forwarded to the closest participant in the finger table not exceeding that key.
2. That participant checks to see if he is the owner of the destination key, if not he returns the closest entry not exceeding the key value in his finger table to the requester,
3. The participant corresponding to this entry is then queried, and this process is continued until the participant who owns the key requested is found.

The guarantee for finding the key requested using this algorithm is given in the strict organization of the key range and finger table entries. Furthermore, this also guarantee that a lookup of any given node in the ring will take no more than Log(N) hops, where N is the maximum allowed members in the ring. By the randomness property of consistent hashing, each participating node is likely to be evenly distributed across the ring, providing optimal placement for the lookup algorithm. For formal proof of this algorithm and its properties we refer to the original Chord specification [20].

4.2.2.1.4 Message handling

On top of this lookup service we provide two types of communication, both broadcast and one-to-one. Both modes again provide two types of messaging objects, serialized objects or files. Point-to-point communication, include the same steps as the algorithm described above. For broadcasting messages through the peer-to-peer network we use a gossiping data dissemination inspired approach.

This type of communication protocol is more commonly used in unstructured peer-to-peer networks, but it is just as well applicable in our case. The way this mechanism works is that the initiating participant broadcasts the message to all distinct entries in its finger table, and each one if these in turn further broadcasts in the same manner. The cut of in this algorithm is when a participant receives a duplicate message, recalled by a unique identifier. It then stops, and does not further issue a broadcast. The proof of this mechanism is left to others [21], but during the evaluation of Javza we experienced quite reliable broadcasting.

4.2.2.1.5 Membership

Entry into the ring is controlled by the same algorithm as for lookups. The key of an entering participant is decided by hashing down his own unique credentials. These credentials could be anything from a Windows GUID and IP/Port address pairs to nicknames. A joining node requests the closest one to this key and by iterations of the algorithm he is forwarded to the owner of that key. The owner's key range is then split into two on the joining participant's key. The predecessor of this node is notified about its new successor in the ring and the join mechanism is complete.

If on the other hand the key of the joining participant is at the exact latter end of the owner's key range, then a split of the key range is not possible, and thus he is denied entry into the ring. To avoid this from happening, we could make the entire set of keys very large. The randomness property of the hashing algorithm would then ensure this, as conflicting hash values are very unlikely to happen.

To exit the ring in a planned manner, the node who wishes to depart, informs its predecessor and successor of this fact. The predecessor updates its successor information to contain the departed nodes successor. The successor of the departing node then inherits that key range. Under normal circumstances not involving failures, the rule dictates that key ranges are always split from and merged to the successor in the ring. The finger table entries of both the predecessor and the successor are also updated to reflect this change in membership. Other participants that have this departed node as a reference in its finger table must proactively discover membership changes.

4.2.2.1.6 Stabilizing Finger tables

In order for the finger tables to remain consistent in the face of membership changes we have implemented a remedy. It consists of a background thread doing lookups on the entries at regular intervals, and updating them to reflect changes.

However, there is a tradeoff to consider here. Updating them too frequently would cause a lot of bandwidth overhead, and strain the peer-to-peer network. On the other hand, infrequent updates would leave the table in an inconsistent state hurting the speed of lookups, especially if posed with a high churn rate. We therefore limit the frequency of updates by using an update control algorithm.

This algorithm executes in a similar manner to how congestion control works in the TCP protocol. The updater thread execution interval starts out at the lower bound of the threshold. It then updates this frequency every time it runs increasing the interval with 0.5 seconds, upward to the upper bound of 20 seconds. Whenever a membership change is noticed, the algorithm drops this frequency down to the lower bound and starts growing again. Changes are noticed through either connection timeouts or lookup mismatch.

Network outages are very common in mobile environments, and there is the possibility that an unplanned departure of the ring happens. Since the finger table entries require the predecessor/successor links as a fallback, these are the two most important links to preserve. Our solution must therefore support two such scenarios, a predecessor leaving or a successor leaving.

In the case of the successor of a given node leaves, the successor’s successor is responsible for absorbing the key range left unused by the departing node. He is responsible for letting the given node know he is the new successor. We find the leaving successor’s successor in the second entry of the finger table if it is up to date. As previously mentioned, we do not support fault tolerance in this prototype and in the event that the finger tables are not updated, the communication fails over. During experimentation we only encountered this situation when removing a large portion of the nodes in within a given range of the ring in rapid succession.

The execution architecture of the communication component is illustrated in subsection 4.2.3.2 in Figure 21. Communication is implemented as both a server and a client. It is listening on a socket for incoming connections as well as providing an app level interface for interacting with the provided services. When a request is inbound, the server creates a new thread for interaction with another node and upon the end of handling this request, the connection is closed.

Although this approach requires a TCP connection for each request, it requires less state to be preserved and is therefore substantially easier to implement and manage. The alternative, an asynchronous implementation maintaining open TCP sockets to peers, would require a lot more effort.

4.2.3 Inter-Process Communication

This subsection details two different implementations of IPC between apps inside the same host. We implement it to avoid Windows's restrictive app sandbox and address the likelihood of apps being suspended while in the midst of communicating. Located in the Arbitrator, it allows apps to adapt to the usage of other apps without the user having to explicitly provoke this behavior. Furthermore, it allows multiple apps to adapt to this usage simultaneously by broadcasting data.

The first and initial approach uses the disk for communication to avoid the sandboxing and provide persistent decoupled communication. This was developed for the release of Windows 8 Developer Preview in September of 2011. The second type is the current IPC deployed in Javza. It accomplishes the same, using tools that were released in the Windows 8 Consumer Preview at the end of February 2012.

Based on the restrictions of Windows 8 our requirements to an IPC mechanism in the app environment can be summarized in the following steps:

1. It should be able to send structured data across the connections, to increase interoperability and simplify consumption of data across different types of apps.
2. The communication should be bidirectional; the mechanism should be able to send data in both directions.
3. Connections should be decoupled, allowing communication to persist across process instances, to support app suspension and resume.
4. Communication mechanism should be stream based. The messages should be conveyed in the manner they were inserted into the connection.
5. Apps should be able to broadcast messages to other apps, in order to make it consumable by multiple apps simultaneously.

4.2.3.1 Disk based

The first approach, as the title suggests, involves communication relayed by the file system. It is implemented as a contiguous series of messages stored in a file on disk, as illustrated in Figure 20. It is a quite crude way of implementing communication between processes, but to our knowledge, it was the only realistic approach at the time. The main disadvantage using this approach is latency. Each side requires a separate thread performing I/O accesses continuously by polling the disk. If instead given simple intervals for polling this would further hurt latency.

Both Windows Runtime and the .NET platform provide file system APIs for inserting event callbacks which are executed upon changes to the file system. We avoid disk polling by exploiting this functionality. The wrappers implemented on each side of the library allow us to subscribe to an event handler which trigger upon incoming data.

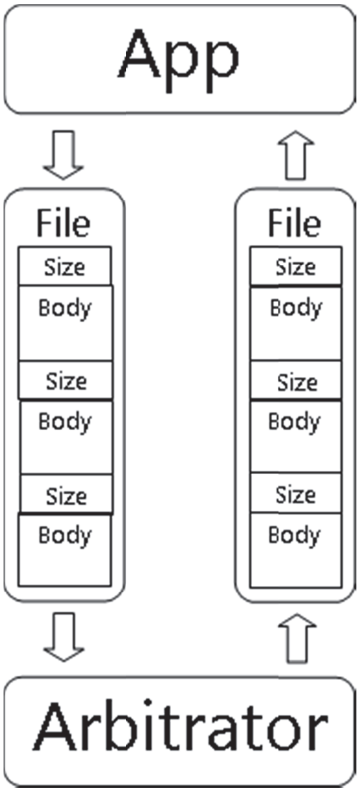


Figure 20: Disk Based IPC

Since file access is exclusive on write, full duplex communication is realized through two single directional files. These provide the logical abstraction of a single bidirectional pipe, as illustrated in Figure 20.

The inner workings that allow the desktop application and the app to use the same files are hidden inside the app's manifest. Capability declarations, as explained in 2.2.2, allow apps to specify storage libraries in which they wish to be allowed access to. With this they declare which file associations they are going to use. So in this implementation each app declares its intent to use the common document folder and its own file format association identical to the name of the app. When the Arbitrator registers the installation of the app it creates two such files, one in each direction, and monitors the one facing the Arbitrator for changes.

The advantage with this approach is that communication is persistent across app states and Operating System transitions. However, the response time and throughput is less than optimal for its usage requirements.

In this implementation the receiver is responsible for removing messages that have been received, meaning he also perform write operations in the same file to delete messages. A consequence of this is lock contention, and we still require further polling for successful requests. These disadvantages are further exposed when running several apps simultaneously.

4.2.3.2 Internet Protocol Stack based

With the Consumer Preview build a new set of developer tools were released, and among these were the “CheckNetIsolation” tool. This tool is meant as a remedy for developers to temporarily disable the isolation of apps from using the loopback address for communication. This enables developers to test their networked apps locally using the local host communication without these restrictions. More specifically, we can use TCP sockets to relay messages to processes inside the same system by addressing the local host IP address (127.0.0.1). We exploit this property and re-implemented the IPC mechanism.

By invoking this tool as a part of the installation procedure, we disable this isolation only for apps installed by our runtime and enable local host communication. Although, each app is still required to add the “Internet Client” capability to its app manifest. This capability simply states its intent to use the Internet and all connected apps require this to function.

Aside from this we still need to address the issue of app suspension. All apps are suspended by the operating system when not executing in full screen. Because of this we cannot assume that the receiving end is active at any point. In turn when the receiving participant is resumed, the sender might be suspended. The disk based approach automatically fixed this, but we are now required to rethink our design. We need the Arbitrator to act as mediation step in communication and save message objects in-flight.

There are two parts to this IPC mechanism and for clarity we will here refer to them as the app side and the desktop side. The desktop side is listening in on a specific message header to initiate communication with the app side. We use the same listener socket as the for the peer-to-peer functionality.

If we recall the app life cycle explained in 2.2.5, apps receive an event from the operating system when they are suspended or resumed. When these occur apps are required to perform an open or close call on the communication channel. The open call on the app side library creates a TCP connection to the desktop side and a separate thread continuously listens for messages on this connection. On the desktop side this connection is managed by a separate thread in the Arbitrator for each app.

When the app is suspended, this connection is closed and the thread resources are relinquished. Messages relayed to suspended apps are put on a queue. When the app then resume and the connection is reestablished, the desktop side forwards these queued up messages to the app.

Both the desktop side and app side threads are blocking on receive calls, and therefore incur little or no extra overhead in resource consumption. We implement our multithreaded capabilities using the .NET platforms Task Parallel Library[18]. It uses a work stealing algorithm to efficiently distribute and schedule tasks across active threads.

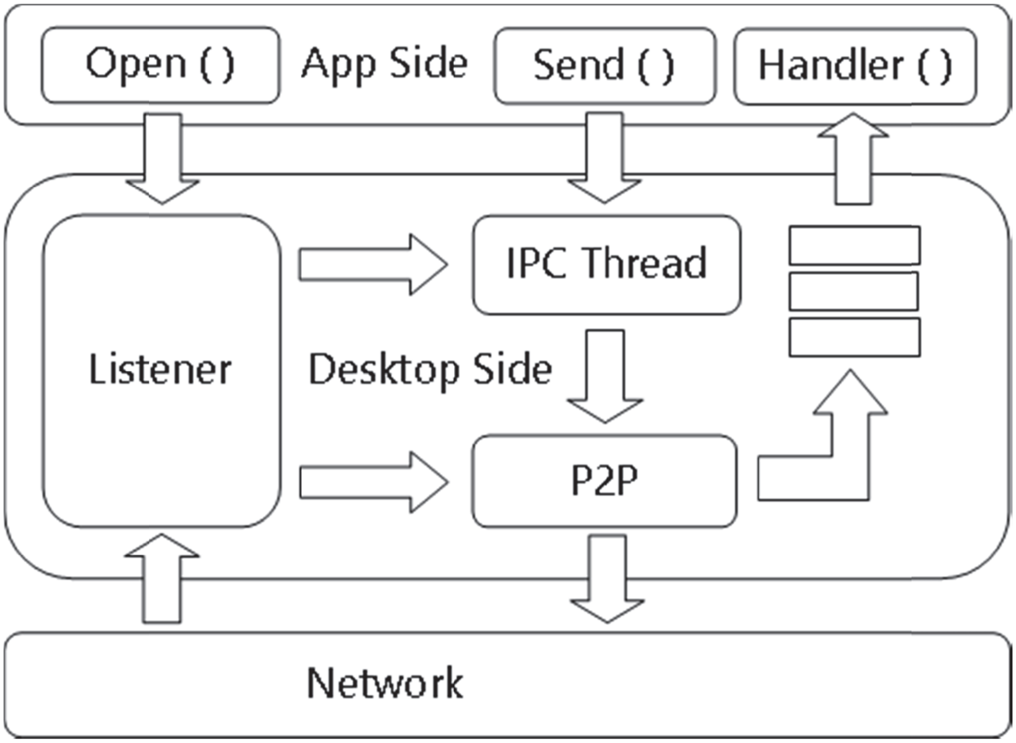


Figure 21: Desktop side and App Side interaction

Figure 21 illustrates the communication between the app side and the desktop side. It also depicts the interaction with the peer-to-peer network. Received messages through the peer-to-peer network are also forwarded through the IPC channel to apps. If the app is suspended theses are also put onto the same queue.

We developed the client side component to conform to the Windows Runtime's paradigm of asynchronous programming by having all inbound messages conveyed through events. This only requires that apps hook on to these events in the same syntactical manner as other event handlers in .NET. The outbound API is implemented as asynchronous "await" able procedures, as in 2.2.8.

This asynchrony as well as the apps volatility due to the app life cycle required us to be much more diligent about how we conveyed messages to the app code. We had to be confident that messages are delivered through the correct event handler, at the correct point in time. We should refrain from delivering them before the app is initialized and ready to receive them.

[18] http://msdn.microsoft.com/en-us/library/dd460717.aspx

Therefore care is taken to ensure this is that each message type only holds a single event handler attached to it at any time. The app is free to change handlers during execution. In the case that no event handler has yet been subscribed to this message type, the library temporarily queues up these messages on the app side. When an event handler is attached these messages are forwarded to it. This implies that apps should only subscribe to events when they are prepared to receive them. Figure 22 depicts the internals of the app side library IPC functionality inside a single app. Each message type has a separate queue and handler associated with it. The messages received are tagged with the type of handler they should be delivered to, and the processing step forwards them accordingly.

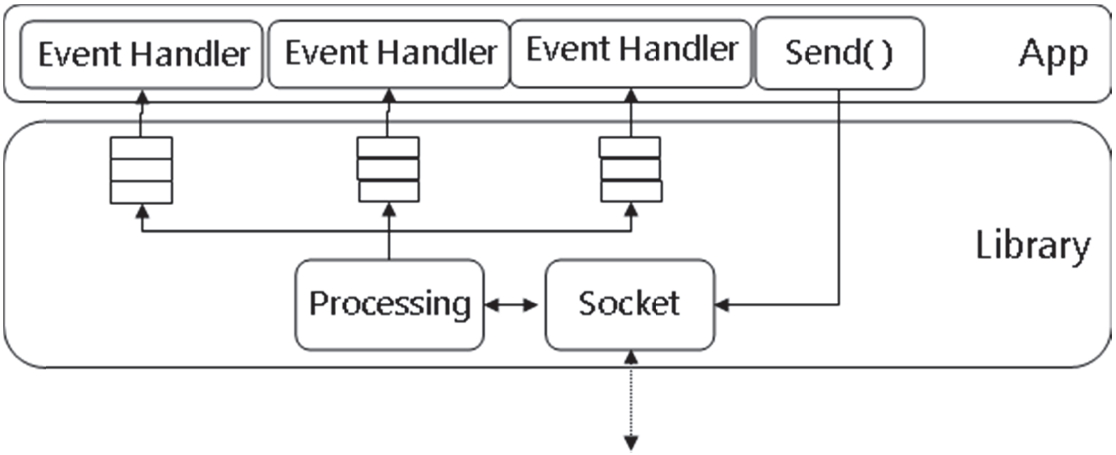


Figure 22: App Side Internals

Event based programming often incurs some synchronization issues which must be explicitly handled by the programmer. Because of this, each of these events are fired in the context of a single thread. This thread is assigned to the communications object upon initialization and changeable during execution. Since most GUI centric apps are not thread safe, this provides a mechanism to allow the event handlers to directly update the GUI objects from the handler code, without tedious interaction with thread dispatchers.

The library is quite extensible, and could easily be modified to support custom event types. The events embedded in the library are specific to the use case we implement in chapter 6 to evaluate our prototype. However, additions to this only yield about 10 lines of extra code, most of which is reusable from the existing event types.

Events of special interest in this environment are the group membership events each app receives when connecting or disconnecting to a peer-to-peer network. Apps can then adapt to what type of communication the Arbitrator can provide at a given time.

4.2.3.3 Naming

The Arbitrator allows apps to transparently communicate with any other app. These could be located on any host in the connected network including its own, both one to one and broadcast. To structure this in a form usable by the Arbitrator for correctly conveying messages we have implemented a naming scheme. The naming conventions are formatted in the following way.

<type>#<username>#<app name>#<custom attributes>

Type denotes if the message being transferred is a file or a generic message. Username denotes the node in the network which holds the app we wish to address. This is the unique credential for each participant in the ring, which is hashed down to the destination key usable by the underlying peer-to-peer network. App name refers to the app name inside a node, de-multiplexed at the receiving end. The last parameter is used in special types of messages, for example files, where this field is used to contain the filename with the proper extension.

Given the case where one wishes to broadcast a message, the corresponding identifier in the naming scheme is left “*” to denote all. In the most extreme case this could be a broadcast to all users and, within them, all apps. If we create an example naming scenario to address all apps on a single host with a generic message, it would look like this:

Message#johndoe##<blank>*

4.3 Summary

In this chapter we explained the implementation of Javza, and what problems each component aims to solve. The resulting prototype evades the restrictions posed by Windows 8 to meet our initial requirements.

We have presented the App Provider, which automatically sends apps to clients for installment. We have presented the GRS, which provides recommendations for users with similar contextual data to join together in groups. Furthermore, we have presented the Arbitrator which manages the client side of the automatic installation procedure and monitors the state of these apps.

This runtime provides communication between apps within the same host and across different hosts. Furthermore, it provides communication across hosts in the group assigned by the GRS through a peer-to-peer network. Each app on every host is now able to communicate with any app on any other host, both one-to-one and broadcast.

5 Evaluation

This Chapter evaluates the performance of Javza, in context with our initial system requirements stated in section 1.2. We conjecture that the system should provide **app integration** both inside and across different host. Furthermore, the system should be **dynamically configurable** allowing apps to be automatically installed based on contextual data. We analyze and reflect on the performance consequences of providing this functionality within the confines of Windows 8.

5.1 Inter-Process Communication

In this section we evaluate the throughput and latency of the Inter-Process Communication (IPC) mechanism for communication between different apps inside the same system implemented in the Arbitrator. To properly evaluate this we need a fair baseline for comparison. We evaluate a system provided mechanism that exposes the same functionality we listed in our requirements (4.2.3). Eligible IPC mechanisms should therefore have the following properties:

1. Should be able to send structured object data across the connection, to increase interoperability and simplify consumption of data across different types of apps.
2. Should be bidirectional, the mechanism should be able to send data in both directions.
3. Connections should be decoupled, allowing communication to persist across process instances, to support app suspension and resume.
4. Communication mechanism should be stream based. The messages should be conveyed in the order they were inserted into the connection.
5. The communication relay should enable broadcasting messages to multiple recipients.

Available IPC mechanisms in Windows include, Named Pipes, Anonymous Pipes, Mailslots, RPC, Shared Memory and COM. Most of these disqualify from this experiment by not fulfilling all our requirements. The only stream based ones are Mailslots, Anonymous Pipes and Named Pipes. Common for them all is that they operate as client-servers.

Mailslots are the only one supporting broadcasting natively, although message sizes broadcast to a domain are limited to 400 bytes. Furthermore, this form of IPC is deprecated and commonly not used by contemporary applications.

Anonymous Pipes are not persistent, and exist only as long as the processes exist. They are not discoverable by naming, and require each participating process to share the system handle to communicate.

As a consequence we chose Named Pipes for our comparative study. Because they are more complex, Named pipes are slightly slower than Anonymous Pipes. However, it supports broadcasting by creating parallel pipes to each receiver without significant overhead. Most importantly, communication is persistent across process terminations, and pipe constructs must be explicitly deleted. Furthermore, it provides bidirectional communication by setting up unidirectional pipes in each direction. To provide the ability to send structured objects across the connection, we use the same approach as with the Arbitrator and add serialization using XML on top of the Named Pipe.

5.1.1 Experimental Setup

The experiments conducted here were exclusively performed on a Dell Precision Workstation 390 running Windows 8 64-bit Consumer Preview build number 8250. The hardware was an Intel Core2 Quad CPU, with 4 cores running at 2.40 GHz with 4 GBs of memory attached to it. All experimental efforts are implemented in C#. Metro apps that are implemented as a part of the experiments use the Windows Runtime API. All other experiments use the .NET framework.

If we compare the two approaches, our Arbitrator requires an extra level of indirection to provide the properties listed above. The experimental setup, as illustrated in Figure 23 and Figure 24, is therefore different for the two types of IPC. In between each process, depending on the communication type, is a series of steps involving interaction with the operating system. We have left it out of the illustrations for clarity.

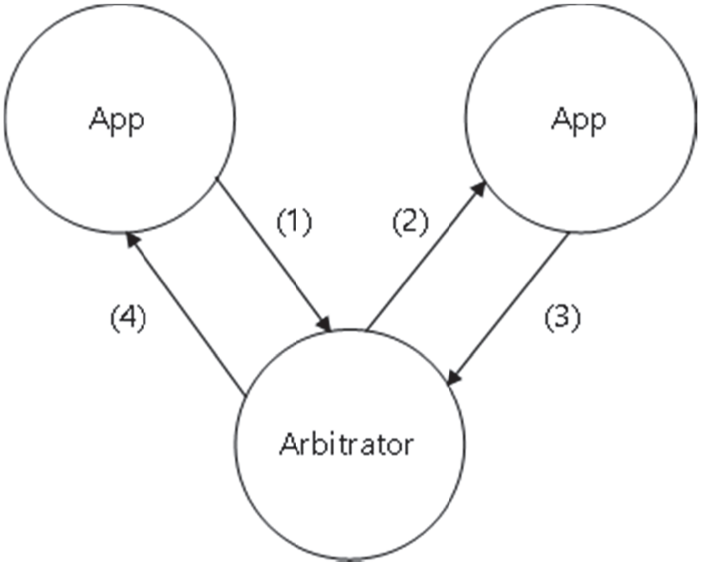


Figure 23: Arbitrator Data Flow

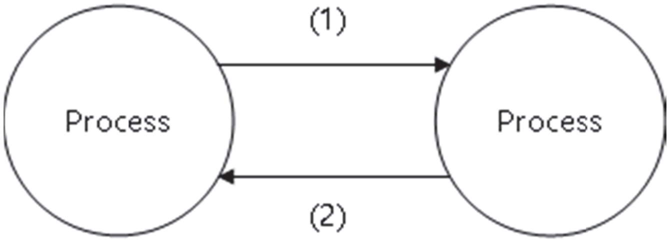


Figure 24: Named Pipes Data Flow

To perform our experiment with the Arbitrator, we require that two apps are running simultaneously. However, in subsection 2.2.5 we explained that Windows 8 is specifically designed against this. Fortunately, the Visual Studio Debugger avoids the suspension of apps running with the debugger attached. This functionality is intended as an aid in developing apps. The developer can instead explicitly trigger app lifecycle transitions, to test how their app responds. We exploit this mechanism and force our apps to remain active throughout the experiment.

Running processes with the debugger attached can have a negative impact on the performance. Since we cannot perform our experiment without it, we require our comparative experiments to run using the debugger as well. We predict that all experiments will incur the same performance loss. However, during our evaluation we experienced little difference in the result.

For each experiment we calculate the arithmetic mean value, or average value, of the collected samples and its associated standard deviation. This gives us an idea of how far from the average measurements the samples are dispersed. We assume that these samples constitute the entire population. The standard deviation for a collection of values is then given by:

$$\sqrt{\frac{\sum(x-\bar{x})^2}{n}}$$

Were $\bar{x}$ is the arithmetic mean of the collection, and n is the collection size.

5.1.2 Throughput

The blue columns on the left in Figure 25 illustrate the average throughput performance of the different IPC mechanisms.

We measure the throughput in megabytes per second to illustrate how much data we can send through a single connection. In the presence of multimedia content it is not uncommon to see large quantities of data transferred across communication constructs. CISCO [22] anticipates that internet video will account for 50 percent of consumer internet traffic in 2012, an increasing amount of which is generated from smart devices. We mirror this trend by setting the size of each message relayed through the connection to 1 megabyte at the core. We then record the speed at which this data is received on the other end.

The small columns in red on the right of each blue column represent the standard deviation of the experiment. On the left hand side we observe that the maximal throughput of the Arbitrator is around 16 megabytes per second. The disk based approach, depicted in the middle column performs at about 1.1 megabytes per second. On the right hand side is the Named Pipe, which exhibits nearly twice the throughput of the Arbitrator, around 30 megabytes per second.

We can observe that the metrics are quite stable and deviate little across experimental tries. The highest deviation is observed for the Arbitrators experiment. This is caused by the other functions of the process affecting its performance, such as reporting back usage statistics. We will give a detailed explanation of the cause in the later subsections.

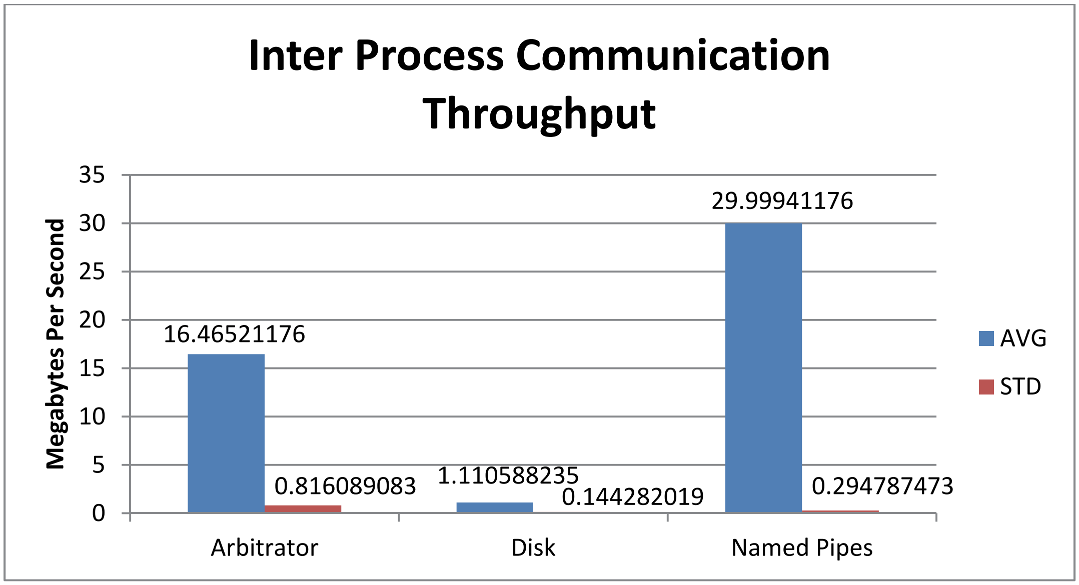


Figure 25: IPC Throughput

An interesting feature of this experiment is that communication handled by the Arbitrator experience throughput of the same magnitude as Named Pipes do, but both experience relatively low performance.

The low throughput is contributed to the serialization/deserialization procedure, which is known to be very costly. Testing without serialization, both local host and Named Pipe throughput remain in the same magnitude only 10x the throughput. Because of the inability to correctly multiplex messages in the Arbitrator without serialization we have only tested this in point-to-point communication for local host.

We can argue that without the indirection posed by the Arbitrator, the performance difference would in fact be even less. The reason for this could be attributed optimizations of the TCP/IP protocol stack inside the operating system. The implementation details of previous versions of Windows suggest this [23].

When a request for communication on a loopback address occurs, the operating system notices this particular IP address in the packets at the IP layer of the network stack. It then short-cuts the connection and avoids the data and link layer. This avoids, among other things, the Network interface card and Nagle's algorithm to buffer management. Nagle's algorithm is a method for improving the efficiency of TCP/IP networks by reducing the number of packages that needs to be transmitted across connections. It delays the transmission and combines outgoing messages until the buffers are full before transmitting the data. The approach also avoids the congestion control mechanism, which could further impact performance. The packages are instead put back onto the input queue up to the transport layer, and delivered to the receiving socket.

Prior to the release of Windows 8 Consumer Preview, the disk based approach was the method of which apps communicated in Javza. Not surprisingly it is by far the slowest in this experiment. The terribly slow seek time of disk accesses are an obvious perpetrator. However, during development and testing we also observed some other interesting characteristics. Most file system employ an exclusive write policy and does not allow multiple processes to write to the same file in parallel. As mentioned in subsection 4.2.3.1, the receiver is responsible for disk cleanup after a message had been received, so both processes actually perform writes to the same file. File access is non-blocking, and the call throws an exception when permission is declined. This requires the participant to retry until success, triggering file access contention even with events instead of polling.

As observed in Figure 25 the average disk communication throughput is about 1.1 megabytes per second. For this experiment we used directly connected pipes in between two separate processes as observed in Figure 24. The sole purpose for including this in the experiment is to provide some insight into the developmental improvements we made during the implementation phase of Javza. Therefore, we did not deem it necessary to include the indirection imposed by the Arbitrator, but rather try to optimize. All reason and logic would suggest that adding the indirection would decrease performance, which also is in keepings with what we observed during our experimentation.

5.1.3 Latency

Response time is vital for many applications, and an important property to test for in IPC is latency. We therefore measure the round-trip time for a message to be transported across the connection, checked for integrity, and back again.

The average rotational latency on hard disk drives is about 4 milliseconds. Adding the file system overhead and contributing factors such as non-sequential reads, this already puts the latency of the disk outside the range of this experiment. For clarity we have therefore excluded it from further experiments.

Figure 26 depicts the resulting average round-trip time in terms of milliseconds for each of the two experimental subjects. In the column on the left we observe that the average round trip time is around 3.4 milliseconds for the Arbitrator. In the column on the right side we observe that named pipes perform at about 0.26 milliseconds, being an order of magnitude faster.

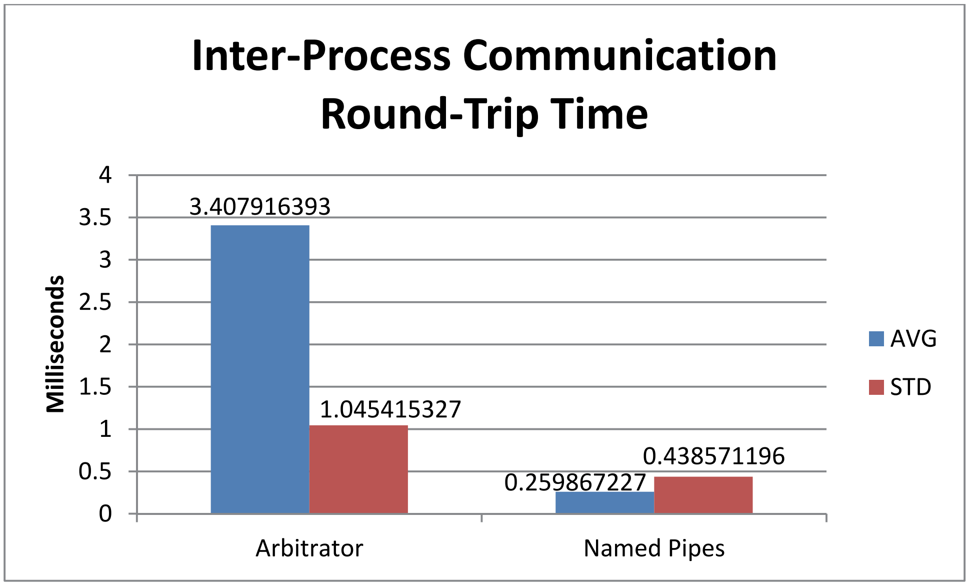


Figure 26: IPC Round-Trip time

The reason for the actual performance difference could be explained by the indirection when transferring messages through a third process, as illustrated in Figure 23.

Each message touches the IP stack, and up again to the Arbitrator process. The Arbitrator process must deserialize the message in order to know where to forward it. After this the message is then serialized, and travel down the IP stack again to be received back at the other app. This app again deserializes the message, checks its integrity and serializes it again. This process is repeated on the way back to the origin. Named pipes do not have this extra level of indirection or extra level of serialization/deserialization and is consequently much faster.

In this experiment we removed 19 outliers in the Arbitrator measurements from a total of 1726 samples, all of which were above 10 milliseconds. Still, we observe that the standard deviation of this experiment is high for both subjects. Since we are operating with timing at such a fine grained level, the timing could be impacted by only the slightest changes in load of the operating system. An example of such could be a page faults occurring. The contributing factor in the Arbitrator is that it simultaneously performs monitoring and report updates back to the App Provider every 2 seconds.

5.2 Inter-Host Communication

We conjure that there are scenarios where apps located on different hosts could benefit from being able to share data. Furthermore, we anticipate that not all scenarios are efficiently servable from a centralized approach to mediate this communication. To enable app integration across hosts without the need for server support, we implemented a peer-to-peer based approach to communication. This was inspired by the Chord internet lookup service [20]. In this section we evaluate the same metrics as in the previous section, but across different host systems.

5.2.1 Experimental Setup

The setup consists of two computers interconnected, running Windows 8, achieving a maximum bandwidth of 100 megabits per second. One of these is the Dell Precision Workstation 390 used in the previous experiments. The other is a Dell Vostro 1500 Laptop with an Intel Core 2 Duo 2.2 GHz processor with 2 GB of memory.

The comparative approach to inter-host communication we use is regular TCP. It is the most optimal and reliable communication mechanism in its segment, available for commodity operating systems. Our peer-to-peer solution is in fact built upon this very mechanism and it should serve as a good baseline for comparison. In order for us to accurately test the round-trip time we must disable Nagle's algorithm for buffer management. This means that TCP packets are sent across the link as fast as they are written to the socket. As with the previous experiments, we require that the comparative communication methods are able to send structured data across the connection. Hence, we serialize data conveyed through the TCP connection.

We initialized two Arbitrator processes, one on each computer, and connected them together in a peer-to-peer network. We wait for the frequency at which finger tables are updated to rise to its upper bound, so it does not infer too much with the experiment.

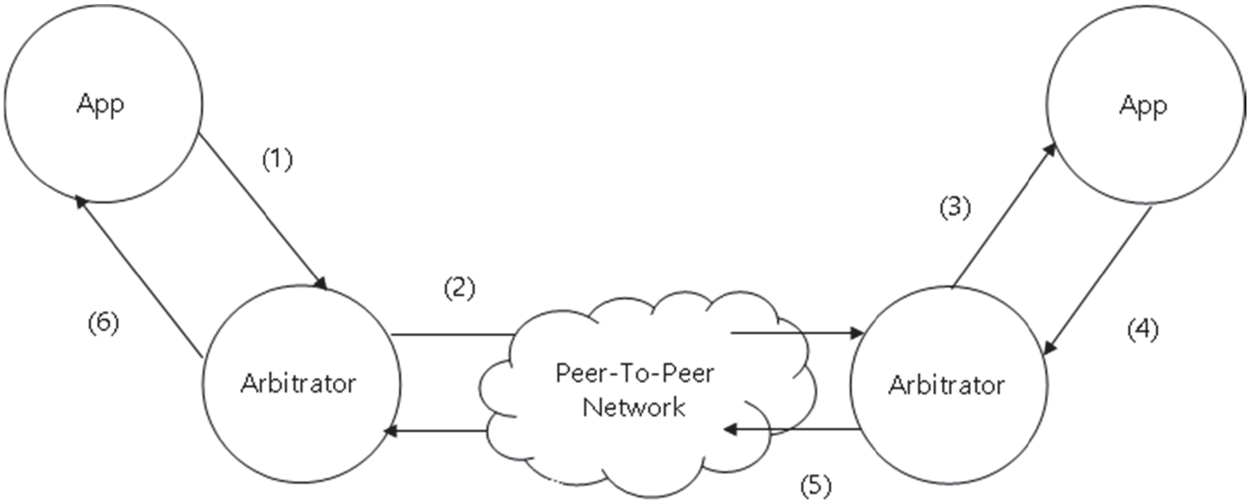


Figure 27: Inter-Host Arbitrator Data Flow

Figure 27 explains the setup of the Arbitrator. In comparison with Figure 28, this illustrates that just as with the IPC experiment the Arbitrator provides extra indirection. However, in this case there are actually two levels.

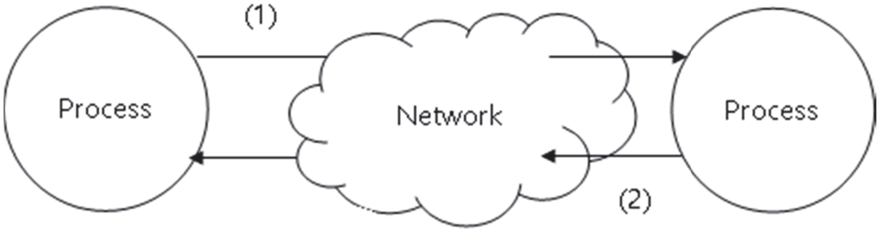


Figure 28: Inter-Host TCP Data Flow

5.2.2 Throughput

In Figure 29, the blue columns on the right hand side of each communication type depict the average throughput. In the left column we observe that the Arbitrator exhibits an average throughput of around 7.9 megabytes per second. On the right hand side we observe that the TCP connection performs similar giving it an average throughput of around 7.2 megabytes per second. Next to these on the left side are the corresponding standard deviation measurements. As with the experiments before, each package we send contains 1 megabyte of data.

We can see that they perform very similar. Since we only add two Arbitrator nodes to this peer-to-peer network, the data only travels one hop. When the app side library forwards the data to the Arbitrator, it directly forwards it to the closest entry to the receiver in its finger table. In our setup the receiver is always the successor node and therefore only one TCP connection is required for each transfer. With further hops in the peer-to-peer network, we anticipate that the throughput will drop.

Another interesting observation is that the Arbitrator slightly outperforms the TCP connection. This is due to the fact that the Arbitrator's peer-to-peer mechanism is multithreaded. It will stress the maximum bandwidth capacity by issuing multiple parallel TCP connections to the receiver. The TCP, on the other hand, is single threaded and send all data across one connection.

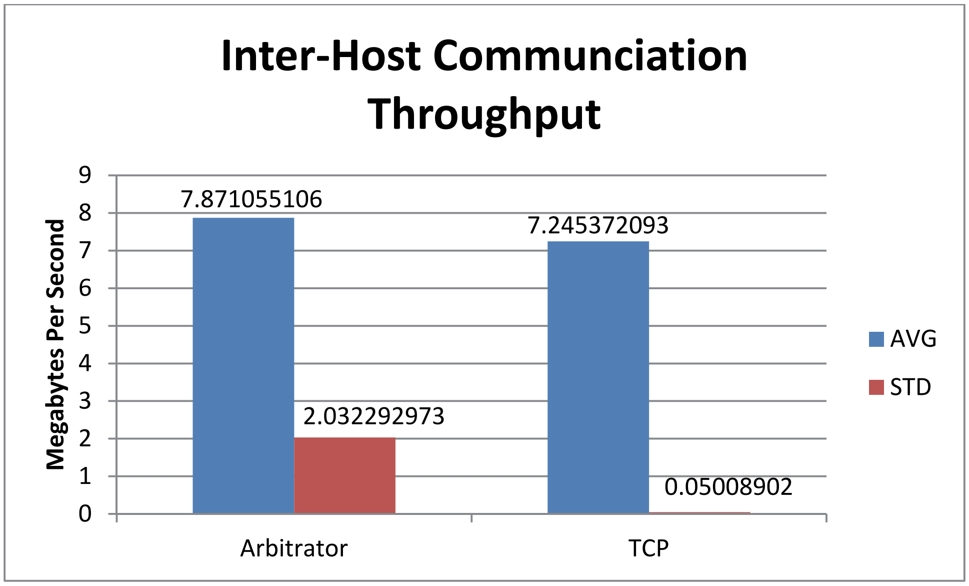


Figure 29: Inter-Host Throughput

5.2.3 Latency

Figure 30 depicts the round-trip time measurements for the Arbitrator and the TCP communication. We observe that the average round-trip time for the Arbitrator is around 8.1 milliseconds. The round-trip time for the TCP connection is around 0.8 milliseconds. This clearly illustrates the overhead of indirection in the Arbitrator nodes, depicted in Figure 27. Since we have an extra level in comparison to the inter-process measurements, the average is about 2.5x over the results gathered from subsection 5.1.3. This corresponds fairly well to IPC time doubling plus the extra overhead associated with the TCP connection across the network (1 millisecond). Without disabling Nagle's algorithm, we experienced a round-trip time of around 400ms for the TCP connection.

In this experiment we removed 6 outlier samples of a total 1727 all above 30 milliseconds for the Arbitrator measurements. However, the standard deviation measurements for the Arbitrator measurements in both inter-host experiments are quite high. During the experimentation we observed that when the finger tables were updating or the usage statistics were reported back to the App Provider, the throughput dropped and the round trip time increased.

The cause of this is lock contention. We restrict finger table access by mutual exclusion from other parts of the Arbitrator while in the midst of updating these. Since the IPC experiments do not interact with the peer-to-peer code, the measurements of these do not exhibit this deviation. We can also observe that the round-trip time is impacted more than the throughput experiment. In that experiment we send 1 megabyte of data for each package so most of the time is spent waiting for sockets to unblock. If on the other hand we were to exchange smaller packages in the throughput experiment, this would impact performance more. This applies to the IPC experiments as well.

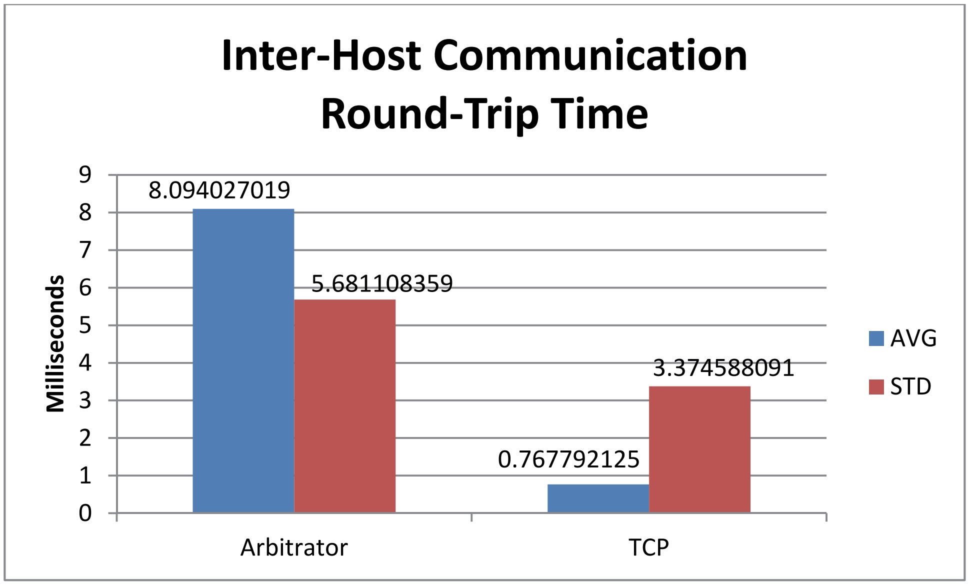


Figure 30: Inter-Host Round-Trip Time

5.2.4 Idle-time overhead

In chapter 4 we described how we bypass the restrictions imposed on apps to provide both inter-processes and inter-host communication. As a result we implemented the Arbitrator as a separate process running outside the Windows Runtime environment. There are consequently some overhead associated with this.

As mentioned in subsection 4.2.2.1.6, the peer-to-peer communication employs an update control algorithm for updating finger table entries in a timely manner. In this experiment we illustrate the idle-time I/O overhead of having the Arbitrator being connected in a peer-to-peer network.

There are several reasons to why we chose to measure I/O resources. First and foremost, since we are restricted to testing on the local computer, we require the metrics to be isolated across processes. Measuring CPU would only make sense if processes did not impact the CPU time available for others. The Windows scheduler does not perform such isolation.

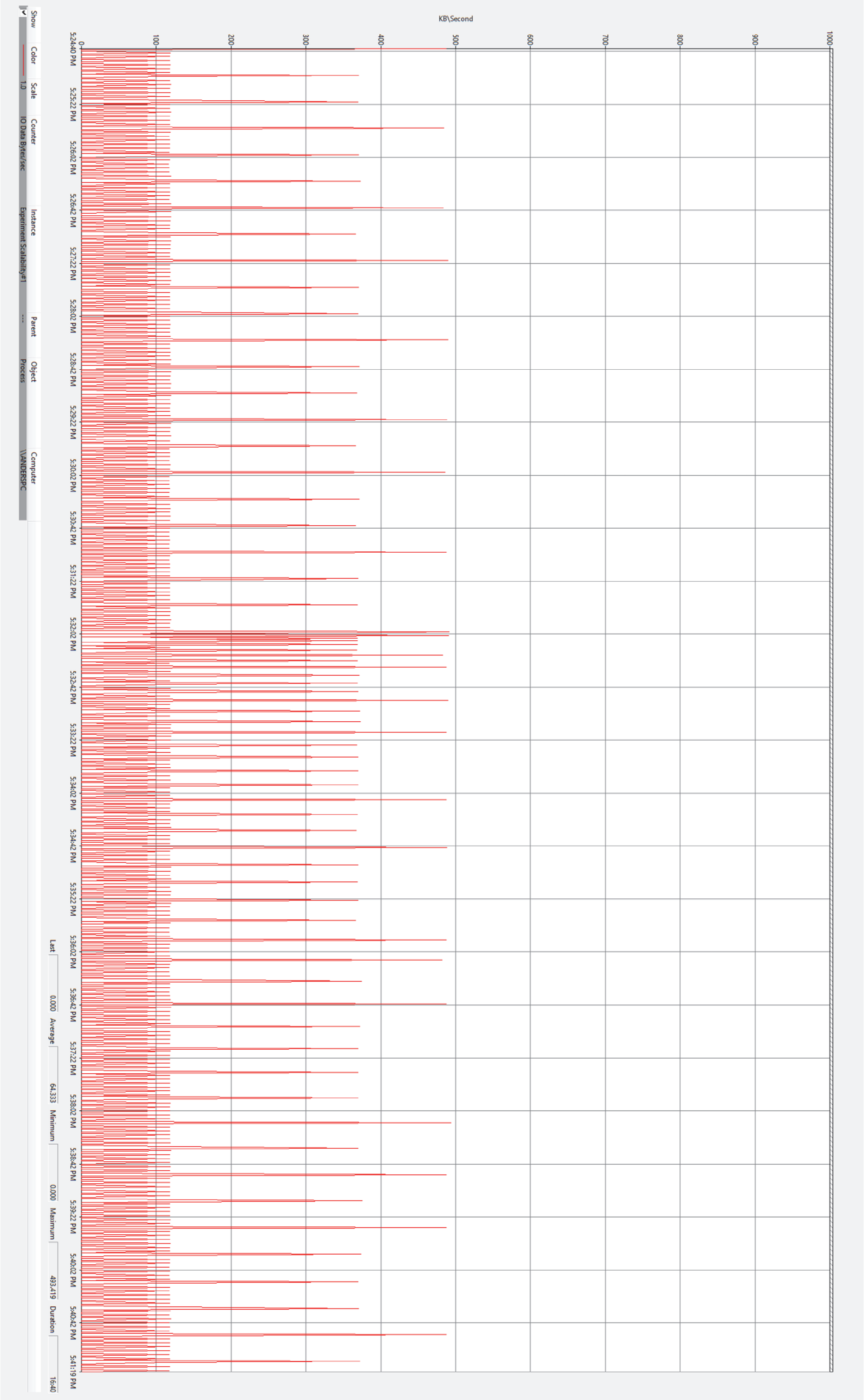
I/O resources are one of the most expensive resources in smart devices, and an important metric to measure in ubiquitous scenarios. Contributing factors to this are real expenses such as metered internet, where internet connectivity is charged by the megabyte. Other expenses include the battery drainage from having the wireless radio link enabled.

Using one computer we start several Arbitrator processes communicating across the loopback interface to illustrate the overhead of being connected to a 39 node peer-to-peer network. Our choice of exactly 39 nodes comes from the fact that it was the largest amount we were able to deploy and connect on the same computer, while maintaining stability in the peer-to-peer network.

We have chosen to monitor a single random Arbitrator instance for performance. Because of the symmetric properties in the peer-to-peer implementation, we expect similar behavior across all. We used Windows own performance monitoring tool called “perfmon” for isolating the performance metrics of a single process and a snapshot of this is depicted in Figure 31. It shows the overhead in terms of total data written or read from I/O per second in kilobytes.

We can be confident that we are only measuring network bandwidth because the only type of I/O performed by our app at idle-time is network I/O. We started 39 instances and connected them together in the exact same way that a recommendation triggers group connection. We wait until the update control algorithm has stabilized before starting the experiment.

The spikes that occur every 20 seconds or so is the finger table update procedure. At this point the frequency is at its upper bound, triggering every 20 seconds. In a stable peer-to-peer network, without membership changes, we only require outbound communication while idle every 20 seconds. The constant cycles below the spikes are triggered by the Arbitrator reporting back app state and responding to update requests from other instances. The vertical range of the graph goes from 0-1000 kilobytes per second, and these contribute about 100 kilobytes of data. The spikes indicating a finger table update are in the range of 300-500 kilobytes per update.



Last	Average	Minimum	Maximum	Duration
0.000	64.333	0.000	493.419	16:40

Show	Color	Scale	Counter	Instance	Parent	Object	Computer
✓	—	1.0	IO Data Bytes/sec	Experiment Scalability#1	---	Process	\\ANDERSPC

Figure 31: Idle Runtime overhead (Total KB/s of I/O)

We now wish to evaluate how the update control algorithm operates in the face of membership changes. To properly trigger the expected behavior we must incur some membership changes to the peer-to-peer network. This was handled by removing 6 participants evenly distributed throughout the ring. By doing this we increase the likelihood of our random Arbitrator instance being affected by the membership change. Just as expected the frequency drops to 500 milliseconds in the middle of the graph. As the finger table stabilizes again, we can observe that the threshold grows back up to the upper bound.

The large quantities of data are caused by XML serialization. Furthermore, all packages are wrapped inside use our own custom message format which incurs more overhead. This message format is on the other hand necessary to address recipients correctly inside the Arbitrator runtime and throughout the peer-to-peer network. An example of how much overhead this incurs could be illustrated by sending a single integer through our message format. This will generate a network load of about 5.6 kilobytes. However, this is only a static cost and includes the whole message header in our custom protocol format.

Despite the overhead of serialization, we can still argument towards the benefit of implementing a threshold based approach to updating finger tables. As observed in this experiment, the frequency of updates is responsive with regards to the peer-to-peer network state. If we were to adopt a more optimized serialization scheme, this would reduce the I/O costs drastically.

5.3 App Installation

In order for Javza to provide dynamic configuration of apps we require the overhead of app installment to be low. We therefore conducted an experiment where we measure the time taken to install an increasing amount of apps simultaneously, using the Windows Package Manager API. The overhead we measure only includes the installation of packages. We exclude the signing and install procedure of the associated certificates.

Figure 32 depicts the total time taken to install a given number of apps at a time on a single computer. The graph illustrates a linear increase in app install time as the concurrent job number increases. We observed in the course of the experiment that installing apps were disk bound. The utilization of the disk was at peak capacity throughout each run of the experiment. The CPU metric, on the other hand, remained within the 15-25% region. Using our initial approach where we installed a separate certificate for each deployment, the time per app increased to about 10s.

Another observation we made during this experiment was that the install process tended to have a warm up phase, making consecutive installations faster than previous ones. The resulting data points are therefore calculated by averages of ten runs across all parameters. We observe that the standard deviation is higher for the middle points in the graph.

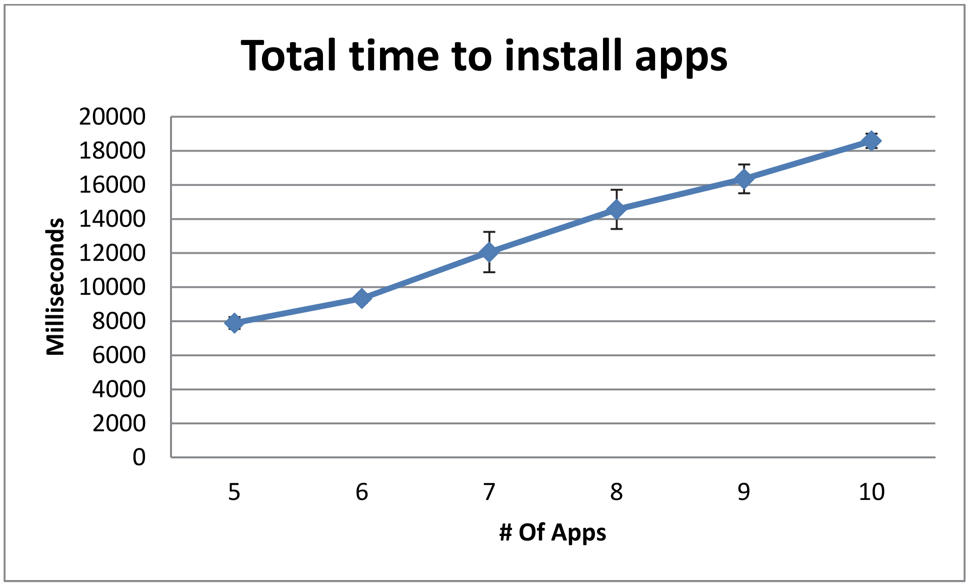


Figure 32: Time taken to install apps

5.4 Conclusion

Our conjecture was that there are benefits to providing integration across apps situated on the same host, as well as across hosts in asymmetric systems. Furthermore, we also suggest that there might be benefits to automatic app configuration and installment. In this chapter we have evaluated this conjecture and the associated cost of supporting it at the client system in Windows 8.

We have conducted experiments on the inter-process as well as inter-host communication implemented in Javza to expose the overhead related with providing app integration. We have further illustrated the cost of dynamically installing apps into this system. We also expose the idle-time overhead of our peer-to-peer service in a connected environment to provide app integration across hosts. Lastly we implemented a use case, which exposes a usable scenario in which an asymmetric system would benefit from our prototype.

In section 5.1 when measuring the IPC performance in terms of common benchmarks, we observed the system-provided Named Pipe outperforms our service. The latency is an order of magnitude slower in our case, and the throughput is cut in half. We contributed much of the extra cost to serialization and communications indirection. If Named Pipes were available for Metro Style Apps this would not be an issue as they satisfy our requirements for communication. On the other hand, we can still conclude by saying that since the apps are seldom operating simultaneously, the latency will not be an issue in real life.

In section 5.2 we observed the cost of using a peer-to-peer based approach for allowing app communication across hosts. The metrics for throughput and round-trip time are affected by providing the services in a separate user level process. We identified the XML serialization as a significant overhead in I/O costs. However, the conceptual frequency modulating algorithm for updating finger tables could provide scalability if serialization is handled correctly.

In section 5.3 we observed the cost of app installation onto Windows 8. We discovered that the process was disk bound, and that the install time scales linearly as the number of installs increase. The time it took for 10 apps simultaneously to complete installation was around 20 seconds.

6 Case study: Collaborative search

The Javza prototype provides dynamic app configuration in Windows 8. Furthermore, it provides app integration by enabling communication between apps residing at the same host and across different hosts. In this chapter we design and implement a case study involving a concrete scenario which utilizes this. The motivation behind this is to evaluate if there exists some scenarios in which smart devices could benefit from our initial conjecture and Javza.

The outline of this chapter is as follows: We first explain the concept of collaborative software and collaborative search. We then explain some of the associated difficulties with this field. Then we design a specific scenario involving a collaborative search experience running on top of Javza. We then explain the implementation of this scenario and how it relates to the Javza architecture. Lastly we will summarize the chapter by highlighting some of the possibilities and constraints with this scenario in the context of Javza.

6.1 Collaborative Search

The term collaboration is defined as a group of individuals working together to achieve a common goal. Collaborative software, or group-ware, is a branch of application software that aims at aiding this process. This type of software is difficult to develop and involves several research disciplines [24].

One needs to understand the psychological and social mechanisms involved in solving a problem in plenum, which is challenging. The social hierarchy of a group of people working together to achieve a common goal is also dynamic in nature and can change as the process continues. Another crucial aspect in developing this type of software is interaction, with both computers and other individuals. This requires understandable and well defined interfaces, which involves interaction design and graphical design.

The scenario we implement is closely related to a concept called collaborative search. In this type of collaboration users are able to cooperate in searching, aiding one another in finding the correct result or goal [25].

There exist two major sub categories of collaborative searches, explicit and implicit collaboration. In explicit collaboration the users share a common agreed upon basis of information and already acknowledge the cooperation. The users then guide each other by performing searches, refining results, and agreeing upon the most relevant information gathered.

The implicit type revolves around users with similar information needs, but not a common ground. The search engine, based on analysis, infers this common interest and provides relevant feedback to each user based on its own analytic work as well as the other users.

A lot of criticism has been dealt to this approach from the fact that it puts the user's privacy at risk. However, if one accepts the premise that collaboration is inherently public in some way, there might still be some benefits from adopting it.

6.2 Scenario

To explain the usage of an asymmetric system model relative to our conjecture we have designed an example scenario. In this scenario we consider a group of people involved in a project. They need to obtain some information regarding this project and in order to do this efficiently they are required to collaborate closely. They realize their information need through search. The required information involves different areas of study, and they must all be researched to satisfy the information need required by the project. For efficiency work is delegated inside the group and each participant is responsible for a given sub category. Although separated, the domains are overlapping, and require everyone to participate and aid one another in achieving the goal.

Each of the participants in this group is in possession of some type of smart device, typically either a tablet or smart phone. Upon initiating their collaborative session, triggered perhaps by a contextual change, they are all connected together into a group. Furthermore, all the apps they require to collaborate are automatically configured into their smart device, each concerned with a separate domain. These apps now have the ability to communicate across smart devices with all apps inside all the other smart devices the group. The participants can now perform their delegated work by the aid of these apps. Furthermore, these apps are also able to share state. By this they enable participants to aid one another in retrieving the information they require.

6.3 Implementation

To model the collaborative search scenario detailed in the previous section, we implement each of the apps to enable search in a separate domain. In an asymmetric system model, as described in chapter 2, lightweight clients are pulling content and functionality from the cloud. To model this we have implemented five Metro Style Apps which retrieve their content from five different cloud services. More specifically each app uses a separate search service to retrieve content pertaining to the query submitted. These five services are listed below in

Table 1.

Service	**Function**
Bing Web Search	Delivers generic web page as the results
Bing Image Search	Delivers images as the result
Bing News Search	Delivers relevant news articles as the result
Amazon Book Search[19]	Delivers purchasable books through Amazon as the result
Spotify Album Search	Delivers album listings from Spotify as the result

Table 1: Cloud service functionality

All of these apps, upon the presence of a search term, interrogates their own cloud service and presents the results to the user in their Graphical User Interface (GUI). Each of these apps are connected to the Arbitrator through the app side library described in subsection 4.2.3.2. Upon activation, they hook on to a special handler implemented for this scenario; we refer to it as the search term handler. Through it they are delivered all terms searched by other apps inside the same host. They then open the IPC pipe which creates a TCP connection with the Arbitrator and activates the IPC enabling them to send and receive data. When each app executes a search by a given term, that term is forwarded to all the other search handlers inside the other apps by help of the Arbitrator. Because the other apps, with high probability, are suspended these terms are queued by the arbitrator and forwarded only when the app resumes execution.

Furthermore, each app also hooks on to a special event handler for detecting group membership. In the event of the host being added to a group, all apps on that host are notified through a special message. When an app receives such a message, it enables extra functionality only available as a part of a group.

Inside the presentation of each app, the list of results from a particular search is displayed. This list further enables the user to click on a particular search result. The app then displays detailed

[19] Amazon authentication mechanism ported to Windows Runtime from C# sample code retrieved from aws.amazon.com

information about this particular result. The click also triggers information about this result to be broadcast through the peer-to-peer network to the same app type on all other hosts inside the group. All apps hook on to a separate handler for receiving this click data, and display the information in a separate column inside the group enabled GUI. Through an identical mechanism presented in a different part of the GUI users are able to share files and communicate by messaging with the same app on the other hosts in the network.

Within the field of collaborative search our scenario can be categorized as explicit collaboration, where all the users share some predefined common ground. However, the collaboration is triggered implicitly by a contextual change in the environment. The contextual change used in Javza is a similar usage of the apps, more specifically, users searching for the same thing. Because of this, we require that each user starts out with the particular app pertaining to the task delegated to them in the collaborative session.

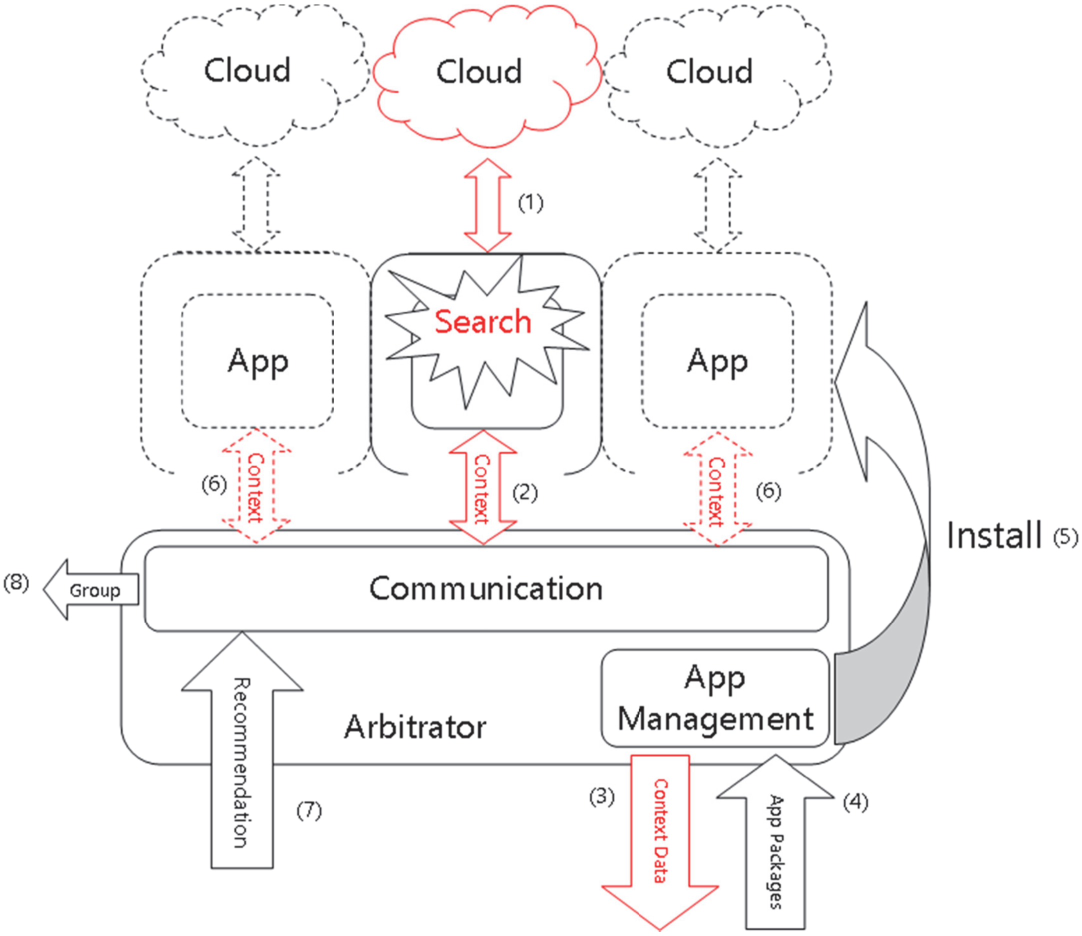


Figure 33: Case study usage

Figure 33 illustrates the Javza system involved in this search collaboration, and how the apps interact with Javza. The App Provider interaction is depicted in the lower right lower corner of the figure, delivering app packages back to the AMC for installation. The Arbitrator retrieves contextual data and usage statistics and sends this to the App Provider. The communication component manages IPC and inter-host communication. It connects to a peer-to-peer network group when a recommendation is received from the GRS. We explain the steps involved in configuring this collaborative search on each participant in the group related to Figure 33:

1. When the single already present on each system app is prompted with a search term, it queries its own cloud service with that search term.
2. The search term is forwarded to the Arbitrator through the IPC mechanism.
3. Upon receipt of the search term the Arbitrator then forwards it back to the App Provider.
4. The App Provider returns the set of apps not installed by this user to the AMC inside the Arbitrator for installment.
5. The apps are then automatically installed onto the client system.
6. Upon completion of the installation, each of the apps have the search term put onto their input queue inside the Arbitrator. When activated by the user they open the IPC pipe through the app side library and hooks on to the search term event handler, thereby receiving the search term. Each app uses this to interact with their own cloud service, presenting the information relevant to their domain.
7. The Group Recommendation Service notices that the participants are searching for the same thing and issues a recommendation to join them together in a group. The Arbitrator instances located on each of these smart devices are then grouped into a peer-to-peer network. The apps are also notified about this event with a special message sent through the IPC mechanism.
8. The apps located on each of these users host system are now, by the help of the Arbitrator, able to share data both within the same host and across hosts in the peer-to-peer network.

Figure 34 depicts all the app tile icons located on the starts screen having been automatically installed as a part of the scenario. The members of the group are now able to see, through each app individually, what types of result the other participants click on.

Inside each app, when a user clicks on an item in the result presentation, a group wide broadcast is issued to the same app installed at all other users. Furthermore, when connected together, each of these apps provide users with the ability to communicate using messaging and share files. The group interaction interface we have implemented for these apps is activated when connected to a group. Figure 34 depicts the group interaction GUI with shared click data displayed on the left and the messaging functionality located on the right.

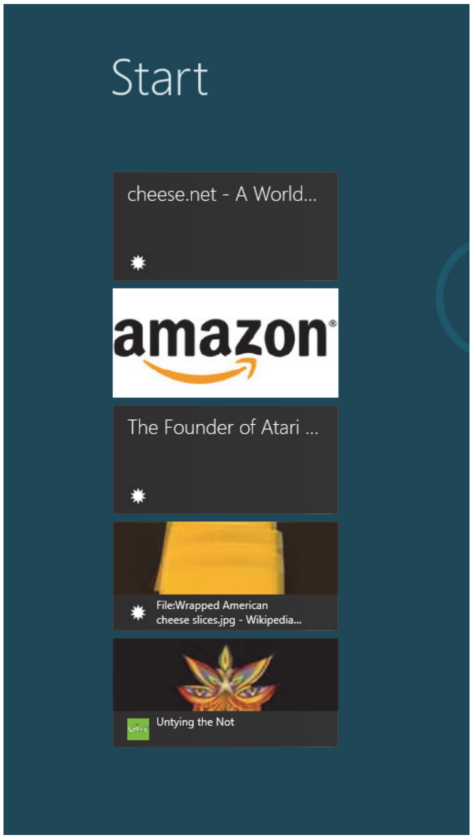
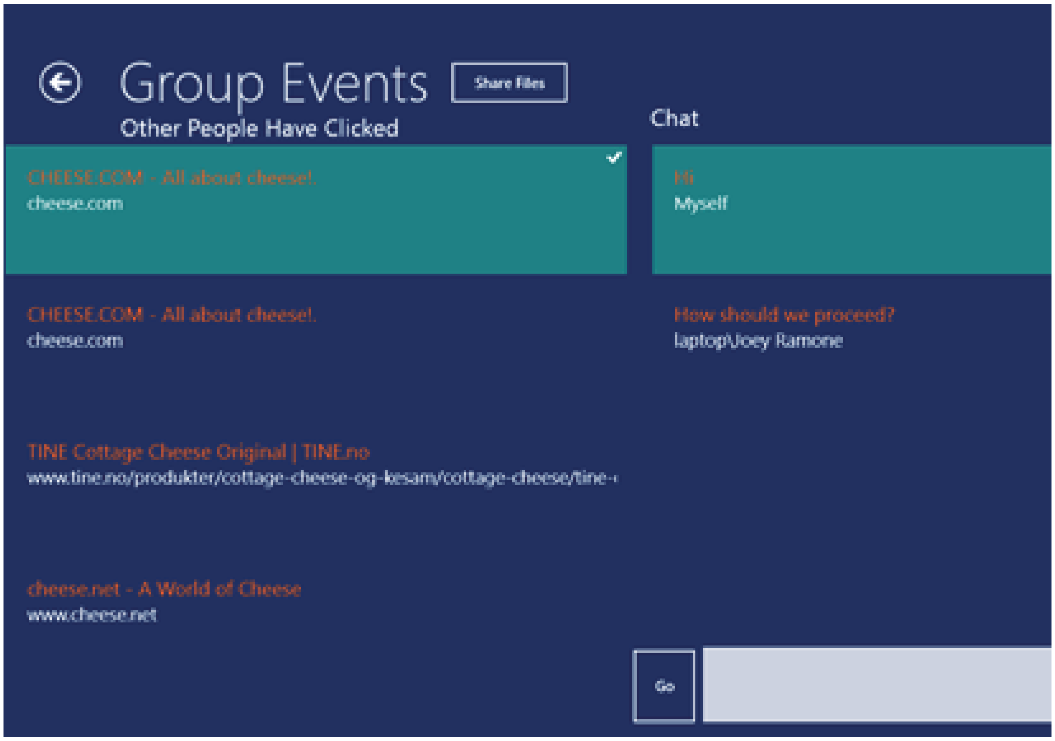


Figure 34: 5 dynamically configured apps displayed on the start screen



CHEESE.COM - All about cheese!.
cheese.com

CHEESE.COM - All about cheese!.
cheese.com

TINE Cottage Cheese Original | TINE.no
www.tine.no/produkter/cottage-cheese-og-kesam/cottage-cheese/tine-

cheese.net - A World of Cheese
www.cheese.net

Hi
Myself

How should we proceed?
laptop\Joey Ramone

Figure 35: A section of the group Interaction GUI inside a single app

6.4 Summary

In this chapter we introduced a collaborative search oriented use case employing our Javza prototype. This enabled automatic installation and configuration of an entire app environment in an asymmetric system environment.

The usage of Javza is not restricted to collaborative search. The apps could function in any type of collaborative domain, using perhaps document editing, computer assisted drawing tools etc. However, this is not only limited to professional collaborative settings. Social settings could benefit from utilizing Javza as well, perhaps involving groups friends playing games. Or perhaps hobbyist organizations working on a project or planning an outing. The only requirement is that the scenario involves a group of users, performing some task which could be aided by smart devices and apps within them.

We argue that without the presence of this functionality, especially in Windows 8, the project collaboration would be more time consuming affair. The participants would have to manually search, download validate and install the apps relevant for their collaboration. Upon this they are required to manually configure a group to communicate with each other, through perhaps a server mediated connection. Once connected, each app on a single device as well as across devices in the collaboration must be explicitly updated throughout the collaboration session to remain consistent. The time consumed in all of these tasks is further extended by the fact that their smart devices have restricted interaction surfaces.

Although a centralized server connection is a viable alternative, one can argue that not all collaborations are organized in the presence of a third party server. Ad hoc collaboration could occur in any mobile environment where smart devices are a realistic choice as aid. In these settings a centralized approach is out of the question. As in situations with restricted bandwidth or metered internet. In these scenarios an ad hoc connected network would perhaps be more fitting. In this case users could instead share the actual apps between them instead of all of them being pushed to devices by a server such as the App Provider. An existing system supporting ad hoc networking between apps on different smart devices will be explained in section 7.3

The issue of privacy is less important within such a group of peers because of the inherent public properties of collaboration. However, communication should be secured from eavesdropping by outsiders. Furthermore, the system should support authentication of membership inside the ring. The security concerns of installing apps automatically require the eligible apps to be conveyed and installed in a secure manner onto the devices. These apps could perhaps be validated by a central authority trusted by the participants, such as the validation procedure present in the Windows Store.

In the scenario presented here, collaboration and configuration is triggered by a change in the environment. Changes could be observed by contextual data retrieved from sensors inside the smart devices. Environmental changes can also mean a change in usage, and in our implementation this change in usage is manifested in the search terms submitted to the apps by users. Another environmental change that could trigger the configuration could be that the GPS coordinates inside the smart devices reveals that certain smart devices are within close range of one another, perhaps in a meeting room.

We realize the collaborative effort without the time consumed in manual app installment. Furthermore, by automatically integrating apps, both within smart devices and inside groups, we remove this overhead as well. The implementation of this scenario is rudimentary, but the design suggests that there might be use cases where the restrictions of Windows 8 will counteract systems design and hurt functionality.

7 Discussion and Related Work

The Javza prototype provides dynamic app configuration and integration of apps both within and across hosts. This functionality is otherwise not present in current app platforms. However, there are several reasons for why these asymmetric systems refrain from providing it. In this chapter we highlight some of the implications of providing automatic app installation and communication. We further explain some suggested modifications applied to the asymmetric system model in Windows 8.

7.1 Implications

The first, and perhaps most serious issue with altering the nature of how apps are installed, is privacy and security in smart devices. These devices handle some of the most private and sensitive data of the typical consumer. These include telephony records, text messages, email, calendar and images to name a few. Anything that might violate or breach this privacy could have severe consequences.

Pushing apps to smart devices for automatic installation could put this at risk. Generally executing downloaded code without proper security mechanisms in place could have potentially disastrous consequences. Systems that provide this capability commonly provide some sort of elevated security for executing this code.

Isolation and elevated security capabilities exist on top of several abstractions in computing systems. System that implement this generally provide an environment for running code in a more secure and less privileged mode, where resources are managed by the underlying system.

Hardware virtual machines execute entire operating systems in guest mode, an example of which is the Xen Hypervisor[20]. The security mechanisms are implemented in hardware providing elevated privilege levels for executing guest operating systems. Privileged instructions trap down to the virtual machine which then executes the procedure on behalf of the calling code. This allows multiple operating systems to run isolated in parallel on a single computer.

Another type of security enforcing mechanism is introduced by the Reference Monitor concept [26]. This is an abstract concept in operating system architectures. A reference monitor mechanism control system access by enforcing an access control policy on subjects, controlling their ability to perform operations on objects inside the system. All access from untrusted subjects to any object is mediated through this monitor. The mechanism requires three properties to be considered a valid reference monitor. The mechanism must always be invoked, as every access must be mediated. If not the case it is possible for an untrusted subject to violate the enforced policy. The mechanism must also be tamper proof, in that it is impossible

[20] http://xen.org/

for an attacker to undermine the security mechanism. The mechanism must be small enough to be able to verify its completeness. Without this last property the mechanism could be flawed.

A related and perhaps overlapping concept to this is sandboxing. The term was first used to describe a method of fault isolation, but is more commonly known for secure isolation of guest programs [27]. These are executed in a tightly controlled environment where access to resources or other parts of the system are prohibited or strictly managed. Guest programs typically only have very restricted resources available, in memory or on disk, for which to run in.

An example system which utilizes sandboxing is available in the Chromium open source browser[21]. Chromium loads web pages in separate processes with a lower integrity level, managed and monitored by the system. These processes allow downloaded scripts to execute, aiding the presentation and functionality of the web page, without security infringements.

Windows 8 provides capability based sandboxing. It executes Metro Style App processes at a lower integrity level, called App Container. Access to system resources is managed via a broker mechanism, governed by the capabilities declared by the app.

What separates the sandboxing in Chrome from Windows 8 is that this type of code execution does not require a rigorous install procedure when executing in the client browser. As mentioned in section 2.2.3 Windows 8 requires app packages to be signed by a trusted certificate before installment. This authenticates the source of the package and validates that it is secure to execute this code inside the client system. Furthermore, capabilities required by the app to function must be validated by the system and the user explicitly.

A substantial amount of research has been put into the concept of verifying code to determine if it is safe to execute. An example of which is proof-carrying code introduced by C. Necula [28]. With this mechanism a host system can with certainty determine if a piece of code supplied by an untrusting party is safe to execute. Untrusted parties deliver attestations as proof that the code adheres to some predefined safety policy. The untrusted party verifies the code by a "theorem prover" which ensures that the code adheres to the policies, and assembles a proof of this. The proof is then packaged together with the code and shipped to the host system. The host system can then rapidly validate the proof, and run the code without any further checks.

If we then assume that we are able to install apps automatically to client environments in a secure manner, there are still other aspects to consider. Users might perceive it intrusive to have apps automatically installed into their smart devices, without asking for user consent. Legal problems arise when we consider a payment model for these types of apps. How is a user expected to pay of an app if he has not explicitly purchased it?

[21] *http://dev.chromium.org*

In Javza we remove the isolation between apps and allow them to communicate with each other. This creates an even broader attacks surface for intruders. The contents of these messages can be sniffed, modified, stolen, replaced or even forged, which can compromise user privacy, and violate security. In [29] the Android platform's IPC mechanism is analyzed in the context of possible security infringements, and several such are found.

In subsection 3.1.2 we explained how we limit the scope of this thesis by abstracting ourselves away from group membership management. We rather declared that all active users are interested in participating in a group. However, we must address the intricacies of dynamically connecting groups of people. There are obvious privacy implications of allowing apps on a user's smart devices to communicate with apps on another smart device. Users should be able to explicitly control their interests. They should be able to manage what groups they are joining and their availability to do so. Furthermore, users should be authorized to be able to participate in these groups and we require authentication to provide the assurance.

Another consequence of our design is resource costs, both in terms of money and power. Providing the capability of automated app installation and cross host integration requires devices to remain connected. Wireless connections are often disabled when not used due to the radio links drainage of the battery. Furthermore, when operating in metered internet environments the actual costs of transferring apps and data is an issue.

Because of the widespread use of Windows, it is also prone to malicious attacks. Throughout its lifetime Windows has been subject to a vast amount of malware attacks, more recently by the flame worm [30] [31]. These types of attacks are common in Windows, while platforms with much less widespread use have been guarded.

Every security mechanism in Windows through its releases had to, more or less, be backwards compatible with existing applications. But with this new app platform Microsoft is now free to redesign the security policies from scratch. It is clear that they are reflecting on past experiences, because they are now taking all precautions possible to guard against malicious attacks. However, this trait does not come without consequences. By doing this they are sacrificing generic and broad support in third party app development for secure and resource optimal apps.

7.2 Changes to Windows

This thesis illustrates that it is possible to implement a dynamic app environment enabling communication between apps. However, as observed, this is not without performance consequences. Properly supporting this with the least amount of overhead would require altering Windows. We will now discuss some propositions for future release of Windows.

First and foremost, installation of apps is as we already have experienced a large performance bottleneck. By making the installation process more lightweight we could remove these. We introduce a concept that would model something closer to what script execution in browsers provide. This approach would emphasize the already present asymmetric property of the app platform. Apps executing inside these smart devices could be mobile and reside on clients only when needed. When not needed, the state and code could be migrated to a cloud service for preservation.

We could analyze the contextual data retrieved from sensors on the smart device to perceive the probable usage. By this we could reason on what apps are probably being used actively in a given contextual situation, and which of them are eligible for migration to the cloud service. As we will see in section 7.3, research has already been done on how to infer this type of app usage.

Another lack in Windows 8 that has caused some commotions in the development community [32] is the lack of Inter-Process Communication. We propose to modify Windows with a mechanism resembling that of Androids intent system. One could even base the solution on top of already existing communication primitives such as named pipes.

The messages could be persisted until received by the other app after reactivation. This would largely benefit the app interested in consuming other app's data, without the burden of having it be done explicitly by the user. This could easily be administered through a broker process, and embedded into the app manifest. Communication could be organized by a publish/subscribe mechanism requiring apps present to subscribe to other apps upon installation, in order to be able to communicate. This mechanism works fine with suspending in place and apps can receive data upon reactivation, if available.

Furthermore, we propose extending this across different hosts. The platform could provide pairing of smart devices into a network. This network could resemble our peer-to-peer approach and enable apps involved on the different devices to share state and data. As we will see in section 7.3, there already exists a framework which provides proximity based pairing of smart devices.

Complementing the asymmetric model in Windows 8 could create a new paradigm in app ecosystems, where each app functions as a part of a dynamic mosaic-like plugin environment. APIs for cooperating are clearly defined and cooperation is optioned after what capabilities an app requires. Apps would function without having to share data, but collaboration happens on a quid-pro-quo basis.

7.3 Related Work

Sohn et al. [33] has performed a diary study on mobile information needs. They conducted a two-week experiment to better understand mobile information needs and how they are addressed. The trials consisted of 20 participants constituting a diverse populous. These were then tasked with recording their mobile information needs, what context they arose, and in what manner they were addressed. They define a mobile environment as being any situation where the normal work station is unavailable. The findings dictated that as much as 42 percent of the time, users postponed addressing their information need, or never addressed them at all. This study provides great insight into what information needs are addressable in given contextual situations. And further what types are addressable, and which should be postponed to a later time where the context is more fitting. In the context of our work, this could prove valuable knowledge in determining what apps to push to a user at a given time.

Microsoft Research in collaboration with the University of Massachusetts, Amherst [34] has implemented a prototype modification to the Windows Phone 7.5 operating system which allows apps to launch faster. The motivation behind it is that app launching on mobile devices is slow and it is not uncommon for these to have a 20 second delay. What they do to remedy this is predict apps launching prior to when they are launched by the user. To perform this prediction they collect contextual data from the abundance of sensors commonly available on mobile devices. The contextual data is fed to a learning algorithm which then computes the decisions on which apps should be started at what time. What they discover from experimenting on this prototype is that app launch times are shaved by about 6 seconds. They further illustrate that the extra strain on battery life incurred by incorporating this modification into the system was just over 1 percent. This work exemplifies another way to adopt learning algorithms to reason about app management decisions in a system. This prototype only involves launching apps already installed, and not automatic app installment. However, this further validates our prototype implementation by illustrating that it is possible to infer what apps a user needs, based solely on the contextual data provided by the smart device sensors.

We previously mentioned the Android intent system, which is similar to what we have implemented here. Another interesting inter-app communication system which is in fact modeled after this is Web Intents [35]. This system provides communication between different web applications. It consists of a lightweight RPC mechanism for communication between applications distributed across the web. It also contains a discovery mechanism, for finding these abilities. This technology is in the process of being adopted by the W3C organization.

A similar approach to our peer-to-peer based communication support has already been introduced in AllJoyn[22]. It is an open-source application development framework developed by

[22] https://www.alljoyn.org/

Qualcomm Innovation Center Inc. It enables ad hoc, Proximity-based device-to-device communication. This supports the development of peer to peer connected apps without requiring cellular networks or internet access. The framework is OS agnostic, and currently in development for Windows 8 Metro Style Apps.

8 Conclusion

This thesis develops and evaluates Javza, a runtime system for dynamic app configuration in an asymmetric system environment, more specifically Windows 8. The runtime supports integration across apps on the same host system, as well as apps receding on different hosts.

8.1 Concluding remarks

Smart devices have introduced new application platforms for implementing apps. These apps are fairly restricted in resources and domain, and they commonly utilize cloud services to provide content and functionality. We refer to this symbiosis between smart devices and cloud services as a new type of asymmetric system. The restricted display space on these devices indicates that all interaction should be kept to a bare minimum. In Windows 8, security and resource considerations complicate app communication both inside and across different hosts. Automatic installation of apps is forbidden for the same reasons. We conjured that allowing this would alleviate much of the interaction necessary for users, and reduce the time consumed in using such smart devices.

We acknowledge that there are several security aspects to consider in a possible solution supporting this. Furthermore, we also consider the privacy and intrusiveness of such a system. However, we still argue that there are benefits to supporting this in app platforms.

8.2 Achievements

The problem statement from section 1.1 is as follows:

"This thesis shall develop and evaluate a runtime system for dynamic application configuration in a concrete asymmetric system environment (Windows 8). This run-time must support integration across applications on the same host, as well as applications receding on different hosts. Alternative solutions with and without server-support must be explored.

The prototype will evaluate important aspects of the prototypes performance. Furthermore we will evaluate the prototype by implementing a use case, more specifically collaborative search. The evaluation will include suggestions for further optimizations and extensions, and possible implications for adopting these."

Within the confines of Windows 8, we have implemented and demonstrated a prototype which supports automatic app installation based on simplified contextual information. Furthermore, our implementation of Javza demonstrates that it is possible to provide integration among apps located at the same host by means of IPC. With the same unified interface we also demonstrated that apps located on different hosts can be integrated in a decentralized peer-to-peer network. We have evaluated the important aspects of Javza's performance, and pinpointed the costs of supporting our conjecture in Windows 8. We have further implemented a functioning use case involving collaborative search to run on top of Javza which demonstrates the possible benefit of our design in an asymmetric systems model.

Lastly, we discussed some of the implications of our conjecture in Windows 8, and provide some opinions on how this could be supported in the future.

8.3 Future work

In this subsection we explain some further improvements that could be made to Javza. Some are design choices, in that we have abstracted ourselves away from the concerns, while others are components that we did not deem crucial to implement in Javza, but should be addressed in future endeavors.

- The communication mechanism used throughout the prototype, is in clear text, and should be encrypted. One way of realizing this is to have the server component implement some type of public key infrastructure.

- Since we already provide a generic API for app integration both internally and across systems, we should provide a mechanism for developers to participate in this and allow them to deploy apps to our App Provider. For this we need an interface for registering apps and usage of these apps.

- Due to time constraints we were unable to fully explore Microsoft’s cloud integration into Windows 8, and we should integrate this into Javza where possible.

- Since no tablets have been released for Windows RT, we have yet to test Javza on this platform. The only way for us to test Javza, have been the x86 version of the system. We want to investigate how to evade Windows RT’s metro-app-only policy, and provide a workaround for this, to allow the Arbitrator to compile to ARM architectures.

- In order to provide relevant apps grouping and user grouping recommendations, we should utilize contextual data retrieved through the sensors on such smart devices. Pending the release of the Microsoft Surface slate, we wish to evaluate our prototype by using the sensor data available in this smart device.

- To limit the scope of this thesis we have not implemented membership detection in our peer-to-peer scenario, but in a realistic case this would be needed. One method of implementing this would be something similar to the gossip based dissemination protocol used in the Arbitrator for broadcasting data.

- Javza does not support being admissible into multiple groups. Groups are assigned on a per-host granular level. We would like to adopt an approach to manage the participation into multiple groups, and be able to seamlessly switch between these.

- Furthermore, we would like to investigate how we could employ the UDDI description and binding mechanisms for web services to couple users with apps. This service couples webs services with consumers on basis of properties described in metadata, and any subject corresponding to the requirements are applicable. We could for example create app requirements descriptions based on contextual information and use a similar approach to match users with apps.

- Since we adopt a non-novel approach to app management and installment, we should investigate what implications this would have on adding a payment model into this. The obtrusive nature of pushing apps instead of requesting them requires us to rethink how we implement this feature. Users cannot be expected to pay for apps that they have not explicitly purchased. A possible solution to this could be delivering apps in a subscription based service.

9 References

1. **Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinsky, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, Matei Zaharia.** *Above the Clouds: A Berkley View of Cloud Computing.* Berkley : Electrical Engeneering and Computer Sciences University Of California at Berkley, 2009. UCB/EECS-2009-28.

2. **CISCO.** Global Cloud Networking Survey. *www.cisco.com.* [Online] 2012. [Cited: 06 15, 2012.] http://www.cisco.com/en/US/solutions/ns1015/global_cloud_survey.html.

3. **International Data Corporation.** Press Release. *www.idc.com.* [Online] 03 28, 2012. [Cited: 06 10, 2012.] http://www.idc.com/getdoc.jsp?containerId=prUS23398412.

4. **D. E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, Paul R. Young.** Computing as a Discipline. *Communications Of The ACM - Volume 32 Issue 1.* 1 1989, pp. 9-23.

5. *Supporting Broad Internet Access to TACOMA.* **Dag Johansen, Robbert van Renesse, Fred B. Schneider.** s.l. : ACM, 1996. EW 7 Proceedings of the 7th workshop on ACM SIGOPS European workshop: Systems support for worldwide applications. pp. 55-58.

6. **PrWeb.** Android Smartphone Activations Reached 331 Million in Q1'2012 Reveals New Device Tracking Database from Signals and Systems Telecom. *prweb.com.* [Online] 05 16, 2012. [Cited: 06 24, 2012.] http://www.prweb.com/releases/2012/5/prweb9514037.htm.

7. **Microsoft Corporation.** Building Windows 8. *Building Windows 8.* [Online] 2011 - 2012. [Cited: 06 26, 2012.] http://blogs.msdn.com/b/b8/.

8. —. Windows Phone Blog - Announcing Windows Phone 8. *windowsteamblog.com.* [Online] 06 20, 2012. [Cited: 06 26, 2012.] http://windowsteamblog.com/windows_phone/b/windowsphone/archive/2012/06/20/announcing-windows-phone-8.aspx.

9. —. Windows Runtime internals: understanding "Hello World" (video) - Build Conference. *channel9.msdn.com.* [Online] 09 16, 2011. [Cited: 07 4, 2012.] http://channel9.msdn.com/Events/BUILD/BUILD2011/PLAT-875T.

10. —. Lap around the Windows Runtime(video) - Build Conference. *http://channel9.msdn.com.* [Online] 09 14, 2011. [Cited: 06 26, 2012.] http://channel9.msdn.com/Events/BUILD/BUILD2011/PLAT-874T.

11. **ECMA International.** *Common Language Runtime Infrastructure (CLI) Partitions 1 to 6 (Ecma-335).* s.l. : ECMA, 2012.

12. **Apple Inc.** App States and Multitasking. *iOS App Programming Guide.* [Online] 03 07, 2012. [Cited: 06 26, 2012.] http://developer.apple.com/library/ios/#DOCUMENTATION/iPhone/Conceptual/iPhoneOSProgrammingGuide/ManagingYourApplicationsFlow/ManagingYourApplicationsFlow.html.

13. **Microsoft Corporation.** White Paper: Introduction to Background Tasks - Guidlines for developers. *Microsoft Developement Center.* [Online] January 27, 2012. [Cited: 06 11, 2012.] http://go.microsoft.com/fwlink/?LinkId=227329.

14. —. Bringing existing C++ code into Metro style apps(video) - Build Conference. *http://channel9.msdn.com.* [Online] 09 14, 2011. [Cited: 06 17, 2012.] http://channel9.msdn.com/Events/BUILD/BUILD2011/TOOL-789C.

15. **Sinofsky, Steven.** Updating live tiles without draining your battery. *Blog: Building Windows 8.* [Online] 11 2, 2011. [Cited: 06 11, 2012.] http://blogs.msdn.com/b/b8/archive/2011/11/02/updating-live-tiles-without-draining-your-battery.aspx.

16. **Mozilla.** Windows 8 Integration. *wiki.mozilla.org.* [Online] 02 8, 2012. [Cited: 06 17, 2012.] https://wiki.mozilla.org/Windows_8_Integration.

17. **G. Salton, A. Wong and C. S. Yang.** A Vector Space Model for Automatic Indexing. *Communications of the ACM.* 11 1975.

18. **MacKay, David J.C.** Chapter 20: An example Inference Task: Clustering. *Information Theory, Inference and Learning Algorithms.* s.l. : Cambridge University Press, 2003.

19. **Fazli Can, Esen A. Ozkarahan.** Concepts and Effectinveness of the Cover-Coefficient-Based Clustering Methodology for Text Databases. *ACM Transactions on Database Systems.* 1990, Vol. 15, 4.

20. **Ion Stoica, Robert Morris, David Liben-Nowell, David R. Krager, M. Frans Kaashoek, Frank Dabek, Hari Balakrishnan.** Chord: A scalable peer-to-peer lookup protocol for internet applications. *Tranactions on networking (ACM/IEEE).* 1, 2003, Vol. 11.

21. **Anne-Marie Kermarrec, Laurent Massouliè, Ayalvadi J. Ganesh.** Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Paralell and Distributed Systems.* 2003, Vol. 14, 3.

22. **CISCO.** Visual Networking Index: Forecast and Methodology, 2010-2015. *www.cisco.com.* [Online] 06 1, 2011. [Cited: 06 15, 2012.]

http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481360_ns827_Networking_Solutions_White_Paper.html.

23. **Microsoft Corporation.** Microsoft Windows Server 2003 TCP/IP Implementation Details. *technet.microsoft.com.* [Online] 06 13, 2006. [Cited: 7 4, 2012.] http://technet.microsoft.com/en-us/library/cc758746(v=ws.10).

24. **Peter H. Carsten, Kjeld Schmidt.** Computer Supported Cooperative Work: New Challenges to Systems Design. *Handbook of Human Factors.* 1998.

25. *Collaborative Exploratory Search.* **Golovchinsky, Jeremy Pickens and Gene.** Boston Massachusetts : s.n., 2007. Workshop on Human-Computer Interaction and Information Retrieval MIT CSAIL, Cambridge, Massachusetts, USA. pp. 21-22.

26. **Anderson, James P.** *Computer Security Technology Planning Study - Section 4.1.1.* s.l. : US Air Force Electronic Systems Division, 1972.

27. *A secure environment for untrusted helper applications confining the Wily Hacker.* **Ian Goldberg, David Wagner, Randi Thomas, Eric A. Brewer.** San Jose, California : USENIX Association Berkeley, 1996. Proceedings of the Sixth USENIX UNIX Security Symposium.

28. *Proof-carrying Code.* **Necula, George C.** 1997. POPL '97 Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 106-119.

29. *Analyzing Inter-Application Communication in Android.* **Erica Chin, Adrienne Proter Felt, Kate Greenwood, David Wagner.** Bethesda, Maryland, USA : ACM, 2011. MobiSys.

30. **Wikipedia.** Flame (malware). *wikipedia.org.* [Online] 06 17, 2012. [Cited: 06 17, 2012.] http://en.wikipedia.org/wiki/Flame_(malware).

31. **Computerworld.** www.computerworld.com. *Computerworld.* [Online] 06 13, 2012. [Cited: 06 17, 2012.] http://www.computerworld.com/s/article/9228064/Microsoft_readies_post_Flame_Windows_Update_changes.

32. **Stack Overflow.** How does the new Windows 8 Runtime (WinRT) compare to Silverlight and WPF? *stackoverflow.com.* [Online] 11 10, 2011. [Cited: 06 17, 2012.] http://stackoverflow.com/questions/7416826/how-does-the-new-windows-8-runtime-winrt-compare-to-silverlight-and-wpf.

33. *A diary study of mobile information needs.* **Timothy Sohn, Kevin A. Li, William G. Griswold, James D. Holland.** Florence, Italy : ACM, 2008. CHI.

34. *Fast App Launching for Mobile Devices Using Predicitive User Context.* **Tingxin Yan, David Chu, Deepak Ganesan, Amand Kansal, Jie Liu.** Low Wood Bay, Lake District, UK : ACM, 2012. MobiSys.

35. **WebIntents.org.** www.webintents.org. *www.webintents.org.* [Online] 06 17, 2012. [Cited: 06 17, 2012.] http://www.webintents.org/.

Appendix A

One CD-ROM containing the Javza prototype source code and experimental data.