UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

Faculty of Science and Technology
Department of Computer Science

Improving Latency in Mobile/Cloud Applications

—

Robert Pettersen

A dissertation for the degree of Philosophiae Doctor – March 2016

Abstract

Smartphones are becoming comparable to desktop computers in terms of computational power, and offer diverse applications ranging from social media and gaming, to multimedia and banking. A particular class of mobile applications, *mobile/cloud applications*, are tightly coupled with the cloud. While executing on the mobile device, they communicate frequently with the cloud for crucial functionality.

Use of cloud-provided services is integral to the operation of mobile/cloud applications. And while the computational power of the cloud is seldom a performance concern, the network latency incurred when connecting a mobile device to the cloud can cause perceptible delays in the application. Users have a low tolerance for delays, so avoiding user-perceived delays is imperative to stop users from switching application providers.

This dissertation presents the Jovaku system, which aims to reduce communication latency between mobile devices and cloud services in a generic way, and by reusing existing infrastructure. Jovaku consists of a middle tier component designed to optimize mobile/cloud interactions and a Software Development Kit (SDK) that allows developers to leverage its capabilities.

The viability of the Jovaku system is substantiated through implementation of several modern mobile/cloud applications. Picster and Dapper both make use of Jovaku to reduce communication latency with their respective cloud services. We also perform an extensive experimental evaluation of Jovaku, revealing latency reduction by as much as 72 % for certain mobile/cloud applications.

Acknowledgements

This dissertation has been made possible by the relentless support and dedication of many proponents. First and foremost, I would like to thank my adviser Professor Dag Johansen, for his advice and feedback on everything from life choices to computer science research.

My colleagues in the iAD group have provided valuable discussions, endless arguments, and ideas that has lead the way for my dissertation. I would like thank Åge Kvalnes for teaching me sophisticated programming skills, by throwing tons and tons of bugs at me, expecting me to fix them. Steffen V. Valvåg for his calm persona and impeccable writing experience, teaching me to write like an adept. Anders Gjerdrum and Håvard D. Johansen have provided precious feedback on the dissertation, keeping my trains of thought on the right track. Erlend H. Graff has been integral to forming the layout of this dissertation with his inexhaustible LaTeX knowledge and continuously high availability.

I would also like to thank the members of the computer science department staff, for making my life as a PhD student most pleasant. I would like to especially thank Ken-Arne Jensen who always seems to have time to hear my complaints and engage in interesting debates. Jan Fuglesteg and Svein Tore Jensen have always taken care of my tedious administrative tasks, leaving me to the interesting research. And of course, the rest of the TK group, Maria W. Hauglann, Jon Ivar Kristiansen and Kai-Even Nilssen, who provided me with technical equipment and free access to the Segway.

Lastly, but perhaps most importantly, I would like to thank my parents, Aud and Jacob, for their unconditional love and support. Without them I would be suffering from severe starvation.

Contents

Abstract i

Acknowledgements iii

List of Figures vii

List of Tables ix

List of Code Listings xi

List of Abbreviations xiii

1 Introduction **1**
- 1.1 Mobile/Cloud Applications . . . 2
- 1.2 Middle Tier Components . . . 3
- 1.3 Thesis Statement . . . 4
- 1.4 Scope and Limitations . . . 5
- 1.5 Methodology . . . 6
- 1.6 Research Context . . . 8
- 1.7 Summary of Contributions . . . 11
- 1.8 Outline . . . 12

2 Background and Related Work **15**
- 2.1 Mobile Device Capabilities . . . 16
- 2.2 Mobile Operating Systems . . . 18
  - 2.2.1 Isolated Execution . . . 19
  - 2.2.2 Application Runtime . . . 21
- 2.3 Cloud Services . . . 24
- 2.4 Developing Mobile/Cloud Applications . . . 27
- 2.5 Middle Tiers . . . 29
- 2.6 Summary . . . 31

3 Optimizing Reads from the Cloud **33**
- 3.1 The Domain Name System . . . 35

3.2 Jovaku Architecture 36
3.3 The Relay-Node 37
3.4 Client Library 45
3.5 Summary 48

4 Optimizing Writes to the Cloud 49
4.1 Extended Architecture 51
4.2 Message Processor 54
4.3 Execution Environment 61
4.4 Client Library 64
4.5 Summary 67

5 Applications 69
5.1 Picster 70
5.2 Dapper 77
5.2.1 User Profile 79
5.2.2 Connecting with Friends 81
5.2.3 Status Updates 82
5.2.4 Making Progress 83
5.2.5 Experiences and lessons learned 85
5.3 Summary 88

6 Experimental Evaluation 89
6.1 Experimental Setup 90
6.2 DNS caching 91
6.2.1 Baseline Performance 94
6.2.2 Jovaku Performance 95
6.2.3 Jovaku with Alternative DNS Configuration 99
6.3 Black box testing 102
6.4 Satellite execution 104
6.5 Relay-node performance 108
6.6 Summary 110

7 Concluding Remarks 111
7.1 Conclusions 113
7.2 Future Work 113

Bibliography 115

A Publications 127

List of Figures

1.1 Components included in existing mobile/cloud infrastructure. 5

2.1 A mobile device communicating through a Mobile Network Operator (MNO) to gain access to the Internet and various cloud services. 16
2.2 Relative size comparison of an early smart phone to a modern smart phone. 17
2.3 Overview of the Android system architecture. 19
2.4 How the Location Manager is used on Android. 20
2.5 Difference between applications running in a VM and docker. 21
2.6 Lifecycle of an Android application from source to running, both on Dalvik and ART. 22
2.7 Lifecycle of an Windows Phone application. 23
2.8 The Xamarin platform binds native iOS and Android SDKs to the .NET platform. 24
2.9 Illustration of a generic cloud video sharing service. 26
2.10 Example architecture of a mobile/cloud application composed of cloud modules from the Google Cloud Platform. 28
2.11 Potential locations a middle tier can be positioned to augment functionality of a mobile/cloud application. 29

3.1 Database lookups with and without Jovaku. 38
3.2 Overview of the Jovaku architecture. 39

4.1 How satellite execution is applied to eliminate extraneous round-trips between a client and the cloud. 51
4.2 An overview of the extended Jovaku architecture. 52
4.3 Layout of a WCF message containing a mobile function with four database operations. 59
4.4 Layout of the custom message format containing a mobile function with four database operations. 60

5.1 Overview of the Picster social network architecture. 71

5.2 Domain name hierarchy illustrating an event “TIL_vs_TUIL”, with its description, location and member list. 72
5.3 Creating and locating events in Picster. 73
5.4 Domain name hierarchy illustrating the media tree under the “TIL_vs_TUIL” event. 74
5.5 The Picster application, which stores image metadata in a cloud database, using Jovaku for caching. 75
5.6 The Picster web application displaying an image from the TIL_vs_TUIL event, with the number of likes and a list of comments. 76
5.7 Handling user profiles in Dapper. 79
5.8 Searching for friends in Dapper. 81
5.9 The feed shows social updates from the user and its closest friends, and changes to the friend list. 83

6.1 Placement of nodes on world map. 92
6.2 Baseline lookup performance for the DynamoDB service in Ireland using the official Amazon C# SDK. 96
6.3 Distribution of worst-case lookup performance with Jovaku and local DNS servers. 98
6.4 Distribution of worst-case lookup performance with Jovaku and Google Public DNS. 101
6.5 Example communication pattern between mobile device and cloud assumed to be of a request/reply type. 103
6.6 Summary of cloud interactions during various mobile application startup. 104
6.7 Comparison of the default serialization and the custom serialization algorithms, with respect to the size of the resulting byte array. 105
6.8 Locations of nodes involved in the experiment. 105
6.9 Observed mean latency when executing a varying number of cloud database queries with and without satellite execution. The error bars show the standard deviation. 106
6.10 Distribution of latencies when adding a friend to a social network, with and without satellite execution. 107
6.11 Latency per bag-of-queries when increasing the number of clients that concurrently submit mobile functions to a relay-node. 108
6.12 Average CPU consumption and throughput at the relay-node when increasing the number of concurrent clients that submit mobile functions. 109

List of Tables

3.1 Layout of the DynamoDB table. 42

5.1 The *Profiles* table contains profiles for users of Dapper. . . . 78
5.2 The *Friends* table contains friend relationships and pending friend requests. 78
5.3 The *Feed* table contains all social updates pertaining to a user. 78

6.1 Machine types used throughout the experimental evaluation, along with labels used to reference them. 91
6.2 Machines involved in evaluating the effect of DNS caching. . 92
6.3 Baseline lookup performance for the DynamoDB service in Ireland. 95
6.4 Lookup performance using Jovaku with the DynamoDB service in Ireland. 97
6.5 Lookup performance using Jovaku with the DynamoDB service in Ireland and Google Public DNS. 100

List of Code Listings

3.1 The interface that needs to be implemented to create a DLZ driver. . . . 40
3.2 Callbacks provided by BIND, for communicating results from the DLZ driver back to BIND. . . . 41
3.3 Example DNS update transaction performed with nsupdate on the *jovaku.com* domain. . . . 43
3.4 The body of the JSON request for the SOA and NS records. . 44
3.5 The header of an HTTP *query* request to Amazon DynamoDB. 45
3.6 The body of the JSON request for updating the value of the *x.jovaku.com* TXT label to "Updated Data Value". . . . 46
3.7 The C# version of the Jovaku programming API. . . . 47

4.1 Interface that must be implemented by mobile functions. . . 53
4.2 Excerpt of the API for accessing cloud-side resources from a mobile function. . . . 53
4.3 Interface for initializing the execution server with a specific underlying implementation. . . . 55
4.4 The APM implementation of the execution server. . . . 56
4.5 The TAP implementation of the execution server. . . . 57
4.6 The WCF implementation of the execution server. . . . 58
4.7 Creating a new application domain, with minimal permissions and set of trusted assemblies. . . . 62
4.8 Excerpt from the IContext implementation used in the isolated application domains. . . . 62
4.9 The sandbox, isolating loading of untrusted assemblies, and execution of code. . . . 63
4.10 Client side interface to utilize Satellite Execution. . . . 65
4.11 Example implementation of a mobile function that provides a bag-of-queries abstraction. . . . 66
4.12 A custom serialization algorithm for a collection type containing strings. First the number of strings are stored, before the strings are added. . . . 67

5.1 The `ProfileUpdate` mobile function will update a user profile, or create a new one if the profile does not exist. 80
5.2 The `FriendRequest` mobile function will insert a new pending friend request into the *Friends* table. 82
5.3 The `StatusUpdate` mobile function will update a user's status by posting the new status to relevant feeds. 84
5.4 The `RetrieveUpdates` mobile function will retrieve social updates and pending friend requests. 85
5.5 The `FriendRequest` mobile function will accept a friend. . . 86
5.6 Comparison of posting status updates to Dapper using Jovaku and using the Facbook API to post updates to Facebook. . . . 87

List of Abbreviations

API	Application Programming Interface
APK	Android Application Package
APM	Asynchronous Programming Model
ART	Android Runtime
BIND	Berkeley Internet Name Domain
CDN	Content Distribution Network
CIL	Common Intermediate Language
CPU	Central Processing Unit
DEX	Dalvik Executable
DHCP	Dynamic Host Configuration Protocol
DHT	Distributed Hash Table
DLZ	Dynamically Loadable Zone
DNS	Domain Name System
EAP	Event-Based Asynchronous Pattern
GPS	Global Positioning System
HTTP	Hyper Text Transport Protocol
I/O	Input Output

IAAS Infrastructure as a Service

iAD Information Access Disruptions

IP Internet Protocol

JAR Java Archive

JIT Just-In-Time

JSON JavaScript Object Notation

JVM Java Virtual Machine

MDIL Machine Dependent Intermediate Language

MNO Mobile Network Operator

NFC Near Field Communication

NS Name Server

ODEX Optimized Dalvik Executable

OS Operating System

PAAS Platform as a Service

PDA Personal Digital Assistant

RR Resouce Record

RS Record Set

SDK Software Development Kit

SFI Centre for Research-based Innovation

SGX Software Guard Extensions

SOA Start of Authority

TAP Task Asynchronous Pattern

TCP Transmission Control Protocol

TPL Task Parallel Library

TTL Time To Live

URI Uniform Resource Identifier

VM Virtual Machine

WCF Windows Communication Foundation

WinRT Windows Runtime

XML Extensible Markup Language

1 Introduction

With the advent of the smartphone, our mobile phone is becoming our personal assistant, providing satellite navigation, fitness measurements, multimedia players, and an alternative to traditional credit cards for payment in shops. Smartphones embed a plethora of sensors, run advanced Operating Systems (OSs), and are becoming comparable to desktop computers in terms of computational power.

Smartphones have diverse applications, ranging from social media and gaming to multimedia and banking. For additional feature enrichment, they commonly communicate with various cloud services. The purpose of these supporting cloud services is typically to make user environments available across devices, so that users can move seamlessly from one device to another, or to integrate external data sources such as news feeds or geographical data. The highly available and reliable nature of the cloud thus complements the roaming and transient nature of smartphones.

Over the last years we have seen the cloud [1] evolve from an Infrastructure as a Service (IAAS) model, providing mostly Virtual Machine (VM) based solutions, to a more fine grained Platform as a Service (PAAS) model, where developers can pick and choose ready-to-use cloud modules to build custom cloud services. These modules include user management and authentication, analytics, and other frameworks for large scale computations that draw on virtually unlimited computational power and storage space. As such, development of new cloud services aimed at supporting mobile applications is greatly simplified.

When mobile applications become tightly coupled with cloud services, a new class of applications emerges: *mobile/cloud applications* [2]. These applications execute on a mobile device, but communicate frequently with the cloud for crucial functionality. This can include locating peers in a social network, retrieving status updates, saving state for migration between mobile devices, or retrieving advertisements. Usually, mobile/cloud applications are inoperable when the cloud is unavailable, but some have support for offline mode, so the application can be used with limited functionality on planes or in other scenarios where the network is unavailable.

1.1 Mobile/Cloud Applications

Use of cloud-provided services is integral to the operation of mobile/cloud applications. When these services are accessed, the cloud-side execution of individual service requests is seldom a performance concern; rather, the main concern is typically the mobile network that is part of the communication path between the application and the cloud [3]. Even on modern 4G networks, the latency—as compared to wired networks—might be the cause of user-perceived delays in the application [4]. Users have a low tolerance for delays, so this may lead to loss of revenue if the customer chooses another application provider [5, 6, 7].

Several well-known techniques can be employed to reduce latency from a device point of view. This includes displaying local animations, hiding latency through background loading or prefetching, employing parallel connections on multiple threads, and caching [8, 9, 10]. Most programming languages for mobile applications provide asynchronous programming abstractions to ease the implementation of these techniques [11, 12, 13].

There are also techniques for reducing the average end-to-end delay in the cloud service itself. By utilizing machine learning and adaptive priorities based on when the request was initiated, the average end-to-end delay can be masked from being user-perceived [14].

A common motivation for mobile/cloud applications is to simplify the client-side application logic by leveraging the availability and reliability of the underlying cloud services. In particular, cloud databases can simplify application logic by serving as highly available repositories for critical state. For improved scalability and availability, these databases are commonly NoSQL [15, 16, 17], with limited support for tabular relations and transactions. This entails a more relaxed consistency model than a conventional relational database. Queries are issued through a programmatic interface, rather than a domain-specific, high-level

query language.

Since NoSQL databases typically lack tabular relations, situations arise where multiple queries have to be executed in a specific order to achieve the same effect as a single query on a relational database [18, 19, 20]. These *dependent queries* require multiple round-trips between the mobile device and the cloud, exacerbating latency issues. Crucially, these cases are hard to mitigate using application-level latency-hiding techniques.

The main focus of this dissertation is on architectural techniques for reducing latency. Specifically, the dissertation focuses on mechanisms for reducing communication latency between mobile/cloud application components. We propose to approach this by introducing a middle-tier component that optimizes interactions between clients and the cloud.

1.2 Middle Tier Components

The concept of introducing a *middle-tier* component—situated between clients and the cloud—has previously been explored in several contexts. For example, code-offloading systems utilize a middle tier to extend the computational resources of a mobile device [21, 22, 23]. There may also be opportunities to conserve energy on the mobile device by introducing a middle tier [24, 25]. These systems might reduce the completion time or energy consumption of certain computations, but they do not aim to reduce the latency of fine-grained operations. Others have tried to augment applications with annotations on methods to increase the granularity at which code-offloading occurs [2].

Conversely, a Content Distribution Network (CDN) can be used to move data closer to the mobile device [26]. CDNs are globally distributed systems of servers pre-populated with data, which can be consumed with low latency due to geographical proximity. One drawback of a CDN is that its main utility is for static content, as a priori knowledge about the distribution is required. When a database is updated frequently, a CDN provides limited value.

When considering database operations, there are two main categories: reads and writes. For reads, a middle tier needs to move data closer to the clients so they can benefit from caching. Generally, writes cannot be cached, but dependent queries that include writes should often be executed close to the cloud database. This leads to a challenging scenario of conflicting concerns for the middle tier: the desire to both move data closer to the client for reading, and move queries closer to the database for execution.

1.3 Thesis Statement

Considering the variety of mobile/cloud applications that will benefit from a middle tier that reduces cloud communication latency, the tier should be *generic* and easily exploitable across applications. Also, applications with a world-wide user base should be accommodated, which implies that the tier needs to span a global infrastructure to cover all possible deployments. Creating a new infrastructure that meets these two requirements is a complex architectural challenge, and practically infeasible. Therefore, a tier that exploits *existing infrastructure* is highly desirable.

Our intuition is that existing infrastructure can indeed be exploited and adapted into a generic middle tier that meets these seemingly conflicting requirements. This could result in a significant reduction in communication latency between mobile applications and the cloud. In short, the *thesis of this dissertation* is that:

A generic middle tier can leverage existing infrastructure to reduce latency for mobile/cloud applications.

To evaluate this thesis, we first aim to design a middle-tier architecture that can be reused across mobile/cloud applications to reduce latency. In the spirit of modular cloud services, the architecture should be straightforward to exploit for both new and existing mobile/cloud applications.

For further evaluation, we implement an instance of the architecture that targets actual mobile devices operating on mobile networks. To create a realistic testbed for the architecture, we target the Windows Phone platform. We develop several mobile/cloud applications that offer typical services, and use them to help evaluate potential latency savings from our architecture.

Recognizing the central role of cloud databases, our thesis evaluation will involve interfacing with existing cloud database services, and access to these services will serve as our primary use case. Because cloud databases are accessed through a well defined Application Programming Interface (API), the usage pattern is similar to any other cloud service that exposes a public API. Results are therefore likely to be applicable to any cloud service that has an API.

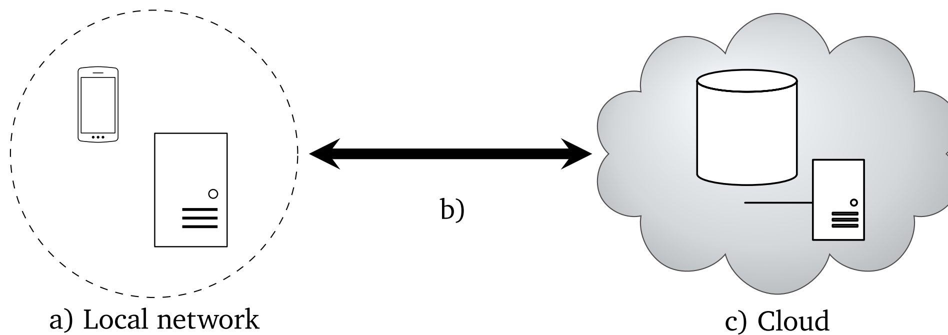


Figure 1.1: Components included in existing infrastructure for mobile/cloud applications. a) refers to components in the local network, such as DNS and DHCP, b) is the communication path between the local network and the cloud, and c) is components situated in the cloud, such as database or VMs.

1.4 Scope and Limitations

Throughout this dissertation, we make certain assumptions about the mobile/-cloud environment and the problem domain, both to focus our attention on the thesis statement, and to guide our design choices when implementing software artifacts. We document these assumptions here, and define the scope for our research by specifying limitations.

- We define existing infrastructure for mobile/cloud applications to include both components in the local network and components in the cloud. Figure 1.1 illustrates these components. In the local network (*a)*) we find necessary infrastructure to communicate on the Internet, such as Dynamic Host Configuration Protocol (DHCP) and Domain Name System (DNS). In the cloud (*c)*) we find backend services to support the mobile/cloud application, such as cloud databases and VMs.

- We focus our attention on reducing latency along the communication paths (Figure 1.1 b)) between the mobile device and the cloud in mobile/-cloud applications. Reducing latency either by improving the application code, or by optimizing the cloud service, will not be considered.

- We adopt the *fail-stop* failure model [27]. In other words, we make the common assumptions that *(1)* processors will halt on failure, rather than make erroneous state transformations, and *(2)* processors can detect when other processors have failed. This allows failures to be detected via *pinging*, i.e. by exchanging regular status messages to signify liveness.

- While scalability is an important concern, and can affect latency when

dealing with an increasing amount of clients, we limit our evaluation to small scale deployments. This is for practical reasons, to allow for rapid development and deployment, and experimentation in a controlled environment. Software and configuration changes would take longer to deploy in large scale environments, and experiments would be harder to reproduce, diminishing their scientific value.

1.5 Methodology

In traditional sciences, theory and experimentation usually follow each other closely. The experimental nature of a scientific method usually involves collecting data on natural processes through observation and experimentation. Theory is applied when observations lead to generalization and the forming of hypotheses. Experimentation can then be used to verify of falsify those hypotheses. Progress is made by repeating those patterns, with changing hypothesis formulations and observations.

Within the natural sciences, where computer science is situated [28], the *hypothetico-deductive* model provides an approximative description of the method of scientific inquiry. The model describes the formulation of a hypothesis, followed by deduction of predictions and the design of experiments that either may validate or contradict the hypothesis. A validation corroborates the hypothesis, while contradictions typically lead to discarding or reformulating the hypothesis. This is an iterative process, where the different steps may be revisited multiple times.

The field of computer science is commonly divided into three disciplines, which corresponds to different paradigms for research [29]:

Theory stems from mathematics, and studies objects whose properties and relationships can be clearly defined and reasoned about using logical reasoning. A prime example is the study of algorithms; given sufficiently detailed descriptions, hypotheses about algorithms (such as the hypothesis that a given algorithm will eventually terminate) can be proved using logical reasoning.

Abstraction stems from experimental science, and constructs models based on hypotheses or through inductive reasoning about observable objects or phenomena. The studied objects could be software or hardware components, or the holistic behavior of a complex computer system. The model is evaluated by comparing its predictions to experimentally collected data. Abstraction resembles the scientific disciplines within

natural sciences like biology, physics and chemistry. Their common goal is to construct accurate models of the rules and laws that govern the behavior of observable objects. Accurate models can be used to predict the behavior in circumstances that have not been observed experimentally.

Design stems from engineering, and use a systematic approach to construct systems or devices that solve specific problems. A set of requirements describes the functional and non-functional characteristics of the construct. Next, the system or device is specified, designed and implemented. Finally, the construct is tested to verify that it meets the stated requirements. If not, the process is repeated, refining and improving the end product with each new iteration.

In practice, these disciplines are intertwined [30], and research typically draws upon all three paradigms to varying degrees. The work presented in this dissertation is not of a theoretical nature, but we draw upon much established theory, for example regarding the inherent properties and limitations of distributed systems. We use abstraction to reason about system behavior at a high level and form hypotheses about how that behavior will be affected by architectural changes. Through experiments we check if our high-level model correctly predicted system behavior.

This dissertation focuses on deriving principles underlying the design of complex distributed software systems in order to improve their design and behavior. Within such *systems research* methods are experimental, emphasizing the construction of actual software artifacts to substantiate conclusions. But these artifacts are not static once created. Rather, they are the subject of a process of continuous refinement where experimental insights challenge assumptions and hypotheses, driving both incremental and radical changes to the design and implementation of the artifacts.

The artifacts are first designed through the process described above without considering real world usability to establish a *proof-of-concept* [31]. Not considering real world usability removes certain constraints and allows unhindered research in the field. The proof-of-concept can either lead to further refinement, or even changing the premise for the concept, allowing for real world usability.

Once a proof-of-concept has been established for the artifacts, evidence for *proof-of-applicability* can be gathered through actual real-world implementations. Collecting reactions and performance metrics from real world adoption yields evidence of usability, leading to *proof-of-usability* [32]. However, gathering evidence for proof-of-usability goes beyond the scope of this dissertation.

Empirical measurements are not only used to substantiate and solidify analysis

and conclusions, but are also integral to a process of continuous refinement where practical experiences challenge initial assumptions, perhaps leading to their invalidation or modification. Contributions therefore often consist of generalizations conveying accumulated experience with the objects under study, along with meticulously crafted concrete objects and experimental results that corroborate conclusiveness. Experimental results convey evidence of performance, leading to *proof-of-performance*.

The architecture presented in this dissertation is the result of refinements and accumulation of experience, a concrete implementation demonstrates its viability and provides a proof-of-concept. Experimental results corroborate its claimed properties and provide a proof-of-performance. Actual applications further demonstrate the implementation's applicability and the efficiency of the claimed properties, and provide proof-of-applicability.

1.6 Research Context

This dissertation work has primarily been performed in context of the Information Access Disruptions (iAD) project, a Centre for Research-based Innovation (SFI) funded in part by the Research Council of Norway. iAD was hosted by Microsoft Development Center Norway (2007-2015) and includes multiple other commercial and academic partners: Accenture, Cornell University, Dublin City University, BI Norwegian School of Management, and the universities in Tromsø (UiT), Trondheim (NTNU) and Oslo (UiO). Several other companies were also to a varying degree affiliated with iAD. The dissertation work relates to previous scientific work done in the research group, and to place this dissertation in the correct context a brief overview of the research group and work done in our group will be surveyed.

The main research focus of iAD was on technologies in support of large-scale information access applications. The focus is vertical, ranging from low-level infrastructure software such as OSs and VMs to the business implications of potentially disruptive approaches to information access and management.

Controlling a large distributed system requires full cooperation and support from the OS running on each node involved. Through our work with OS architectures [33] we gained substantial experience with pervasive monitoring and scheduling. The new Omni-Kernel architecture ensures that all resource consumption is measured, attributed to activities utilizing the resources and permits scheduling decisions on a fine-grained level. The viability of the architecture is substantiated through an actual implementation [34] and shows that performance isolation is a viable way to achieve resource control on the

lowest level [35].

Taking the step into the cloud, OSs become integral to provide the foundation for cloud services that autonomously adapt their capacity to workloads over time, allowing consolidation and resource sharing over potentially tens of thousands of worker nodes. Interference from resource sharing can cause unpredictable performance when VMs are consolidated on a single machine. Virtualization has proven consolidation and isolation benefits, but invariably incurs an overhead. This penalty is notable for latency sensitive tasks, such as TCP processing. Several approaches have been investigated to improve the performance of hypervisors serving VMs that require low latency, high performance TCP connections [36, 37]. Significant performance gains were experienced running HTTP benchmarks, approaching native performance for certain workloads.

Above the hypervisors in the vertical architectural stack, we find parallel, distributed algorithms such as the MapReduce [38] style analytical programming model. Our experiences from mobile agents [39, 40, 41] and MapReduce-style distributed data processing [42, 43] have inspired some key aspects of this work. As in Cogset [44], we promote a functional programming model using the visitor pattern, where latency-sensitive code has the ability to *visit* the backend storage service as desired. In this case, a visitor also resembles a mobile agent; although restricted to moving back and forth between a client device and the cloud, it retains the defining ability to carry state. Rusta [45] draws on insights from our previous work in the area of big data analytics, and explores decentralized deployment of cloud services, where clients can offload to the cloud and to other clients.

With Cogset we explored new mechanisms for routing and placement of data in a MapReduce engine. In contrast to conventional MapReduce engine designs, Cogset employed predetermined data routing schemes to avoid the need for temporary copies of intermediate data. This combination of tight coupling of storage and processing and a functional style of programming resulted in better data locality, and, as a consequence, significant performance improvements.

Public cloud offerings providing readily available computation and storage solutions. Enterprises wanting to exploit these offerings must often invest in in-house computation resources, or private clouds to address security and privacy concerns that public clouds are not able to mitigate. Our work with *Balava* [46] addresses these issues by federating multiple clouds, both public and private, where computations involve data with confidentiality constraints. Cross-cloud deployments of Balava proved to be comparable with native Linux performance in terms of throughput.

While Balava considers confidentiality and privacy, our work with *Suorgi* [47] proposes a new way to structure overlay networks spanning multiple clouds. Suorgi introduced the concept of *meta-code*, which extends and augments existing cloud infrastructure to implement fine-level trust policies, replication management, auditing and provenance mechanisms. With meta-code novice users can construct custom private clouds without explicit programming knowledge. Evaluation with a cloud backup mechanism revealed a significant improvement in throughput and completion time when backing up to multiple clouds for redundancy.

As the number of nodes involved in federated cloud overlay networks increases, intrusion tolerance becomes an issue. We addressed this in our work with *Fireflies* [48, 49], which is a scalable protocol for supporting intrusion-tolerant overlay networks. Fireflies can be used to build intrusion-tolerant Distributed Hash Tables (DHTs) or overlay routing networks. Evaluation shows that even with 280 participating nodes, the overhead incurred using Fireflies is low, and we believe that Fireflies will be able to scale up an order of magnitude without difficulty.

The number of cloud nodes involved often increases as the number of users using the cloud service increase. As the number of clients using a service increases, the attack surface of the service will increase accordingly, as the various clients might have vulnerable software. Software security patches contain information about the vulnerabilities, and attackers can reverse engineer these to gain insight into exploits. Building on Fireflies, we built *FirePatch* [50], an intrusion-tolerant dissemination mechanism that combines encryption, replication, and sandboxing such that the window of opportunity for attackers is minimized. FirePatch is highly resilient to omission attacks, and allows for an intrusion-tolerant overlay substrate for disseminating software patches without using trusted mirrors.

Sports analytics is a growing area of interest, and our collaboration with elite soccer club Tromsø IL and the Norwegian national soccer team (men) inspired a host of interesting work. *Bagadus* [51, 52] integrates sensor systems, annotations and video processing to monitor live soccer matches. The system allows tracking of individual players, and provides stitched panorama video summaries. *Muithu* [53, 54] provides coaches with a simple way to annotate live matches, allowing readily available pre-processed video during half-time to improve coach feedback to the players. Further, Muithu provides a social network for the players and the coach to track training and nutrition, optimizing training based on effort and injuries. The privacy of the players was preserved with our work with *Code capabilities* [55], by embedding executable code fragments in cryptographically protected capabilities to enable flexible discretionary access control in the cloud.

Bagadus and Muithu generate high quality video from multiple angles, and distribution of the video to fans and spectators will require great bandwidth. Our work with *DAVVI* [56, 57] not only allows for multi-quality video distribution using torrent technology over HTTP, but enables efficient video searching, personalization and recommendations for interested parties.

The work in this dissertation is an extension to the broad, vertical stack of previous work in the iAD research group. By exploiting our experiences from performance isolation to distributed multimedia applications, we address related performance issues in the context of smart phones connected to mobile networks communicating with the cloud.

1.7 Summary of Contributions

The major contributions of this dissertation are based on work presented in the papers [58, 59, 60, 61] outlined in Appendix A.

Here we give an overall summary of these contributions:

Jovaku

Proof-of-concept We have implemented a programming interface, Jovaku, which allows developers to leverage existing global infrastructure to cache database lookups close to client devices. Further, we have implemented a driver for the BIND DNS server to take advantage of Amazon DynamoDB as a backend for storing DNS records, effectively allowing database lookups through DNS. Not only can this reduce latency for database lookups, but it can also reduce database cost when cost is a function of lookup volume, by reducing the number of database accesses.

Proof-of-applicability We developed a mobile ad hoc geosocial network application, Picster. Picster is a collaborative image sharing application for events where users are in close proximity, which utilizes DNS to store image metadata and user comments for shared images. Picster has been used during soccer matches at Alfheim stadium, and helped reduce overall bandwidth consumption while providing low latency access to images and user comments.

Proof-of-performance We have performed experimental evaluation, measuring the performance of Jovaku from geographically distributed nodes around the world. Further, we provide a surveyed baseline communi-

cation latency for various mobile networks, geographical locations and public DNS providers. We reveal interesting side effects of the standard network stack when dealing with standard HTTP requests. Evaluation reveals that applications benefit from caching in Jovaku even with hit rates as low as 5 % to 10 %.

Satellite Execution

Proof-of-concept We have extended Jovaku with *satellite execution*, a generic cloud execution mechanism that can be instantiated on-demand from mobile/cloud applications. Satellite execution serializes objects and moves them for execution in the cloud, enabling low-latency access to cloud services. This mechanism proved efficient to reduce latency in key/value databases when executing dependent queries, where several queries must be executed in-order to complete a logical transaction.

Proof-of-applicability We developed a modern mobile social networking application, Dapper. Dapper is a social network that leverages satellite execution to allow users to create profiles, connect with friends and share status updates. The application implements the traditional cloud backend service as mobile functions in the application, and leverages satellite execution for low latency access to the cloud database. Dapper has been used by the iAD research group, providing similar functionality as modern social networking applications.

Proof-of-performance We have performed experimental evaluation, measuring the performance of satellite execution. Further, we provide general insights into how existing mobile/cloud applications communicate with the cloud, yielding indications that dependent queries occur in practice. Looking at a concrete implementation of a social networking application, we reduced the completion latency from 450 ms to approximately 125 ms for certain operations.

1.8 Outline

The rest of this dissertation is organized as follows:

Chapter 2 surveys current mobile/cloud platforms and outlines relevant background information.

Chapter 3 describes the Jovaku architecture, which has potential to save

latency when mobile/cloud applications reading static or semi-dynamic content from the cloud.

Chapter 4 describes the satellite execution extension to Jovaku. Satellite execution offers a programmatic way to interact with cloud databases to potentially reduce latency when executing dependent queries.

Chapter 5 presents Picster and Dapper, two mobile/cloud applications leveraging Jovaku to reduce communication latency.

Chapter 6 describes experimental evaluation that shows the extent to which Jovaku can reduce latency for mobile/cloud applications.

Chapter 7 concludes and outlines possible future work.

2

Background and Related Work

The first mobile device with a touch screen was created by IBM in 1995, and had e-mail and Personal Digital Assistant (PDA) features. Since then, many companies have contributed to shape *smartphones* as we know them today, with top models sporting computing resources comparable to those of desktop computers, and boasting an equally wide range of applications.

The most popular smartphone platforms are currently Android from Google, iOS from Apple, and Windows Phone from Microsoft. These platforms all include a customized Operating System (OS) that provides an isolated execution platform for mobile applications, and a distribution channel that developers can use to distribute applications.[1]

Mobile applications are increasingly dependent on cloud services to enhance the user experience and to provide necessary infrastructure. For example, cloud services can be used to save application state, share pictures or social updates, check for program updates, retrieve ads or to augment the computing power of mobile devices. This interplay adds architectural complexity, and introduces new challenges for application developers.

1. Mobile applications are often referred to colloquially as *apps*; for clarity, we avoid this shorthand term.

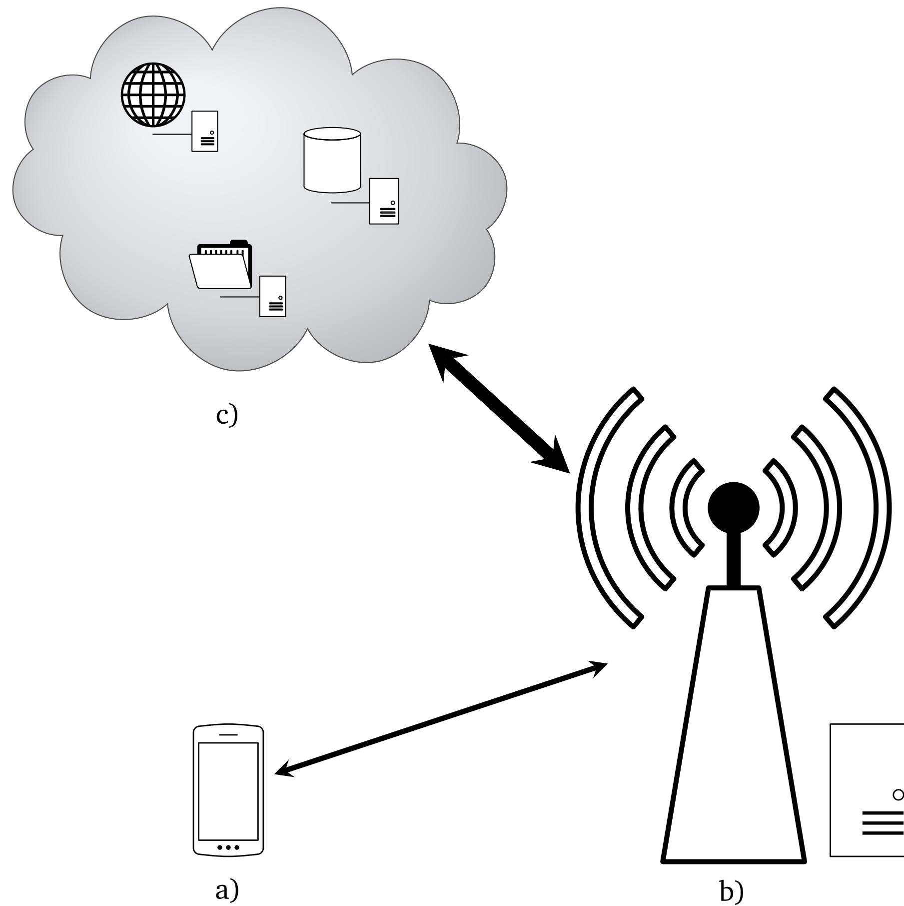


Figure 2.1: A mobile device communicating through a Mobile Network Operator (MNO) to gain access to the Internet and various cloud services.

This chapter outlines the networking and computational capabilities of modern smartphones, explains the differences and commonalities of the most prominent smartphone platforms, and details how mobile applications commonly use and depend on cloud services in their operation. We review general architectures for such mobile/cloud applications and describe how they can be designed and implemented within various programming frameworks.

2.1 Mobile Device Capabilities

Figure 2.1 shows a generic overview of a mobile device and its communication paths to the cloud. *a)* is a mobile device that communicates with the Mobile Network Operator (MNO) in *b)*. Through the MNO, the mobile device

(a) The HTC Dream [62]. **(b)** The Microsoft Lumia 950 XL [63].

Figure 2.2: Relative size comparison of an early smart phone to a modern smart phone.

can communicate with the cloud in *c)*, which hosts a number of different services.

The MNO offers necessary infrastructure services to enable communication to and from the mobile device: as a minimum, the Dynamic Host Configuration Protocol (DHCP) and Domain Name System (DNS) services. DHCP assigns an Internet Protocol (IP) address to the mobile device, and DNS allows the device to translate hostnames into IP addresses. These two services are essential for any device, mobile or stationary, that requires Internet communication. The MNO can support different communication standards that offer varying speeds and latencies. The latest commercial standard is called 4G and allows for theoretical peak speeds up to 1 Gbit/s, but most MNOs offer speeds to consumers in the range of 5 Mbit/s to 150 Mbit/s.

Smartphones have improved greatly in terms of computing power and capabilities since their inception. The first smartphone running Android was the HTC Dream, depicted in Figure 2.2a. The phone was released on October 22, 2008.

It was equipped with a 528 MHz Central Processing Unit (CPU), 192 MB RAM and 256 MB ROM. The display was a 3.2 inch touch screen with a resolution of 320x480 pixels and 64k colors. In addition, there was a 3.15 megapixel camera and an accelerometer sensor.

Figure 2.2b depicts Microsoft's Lumia 950 XL, as an example of a modern smartphone. It is equipped with 2 GHz octa-core CPU, 3 GB RAM and 32 GB internal flash disk, extendable with a microSD card. The display is a 5.7 inch touch screen with a resolution of 2560x1440 pixels, with 16M colors. The camera has a 20 megapixel lens, capable of 4K video recording at a rate of 30 frames/second. The phone is also packed with sensors, including a gyroscope, magnetometer, barometer, compass, proximity sensor, 3D-accelerometer, ambient light sensor, iris scanner, Bluetooth, Global Positioning System (GPS), and Near Field Communication (NFC). These specifications are comparable to modern desktop computers, but the variety and complexity of the hardware impose many new requirements on the OS.

2.2 Mobile Operating Systems

To manage resources efficiently, smartphones have their own OSs, of which Android, iOS, and Windows Phone are the most popular. Figure 2.3 shows an overview of the Android architecture. Android has a modified Linux kernel running at the lowest layer, interfacing with the underlying hardware. This layer hosts drivers for the different hardware sensors and components. While most of the changes to the Linux kernel pertains to access control for driver interfaces, some new functionality has also been added. This includes the new YAFFS2 filesystem for NAND flash drives, a low-memory process killer, and a new power management system [65].

At the next layer, various libraries offer functionality such as cryptography, graphics, and media frameworks. There is also a lightweight standard C library that has a smaller footprint than the GNU C library used in Linux. This layer also contains hardware abstraction libraries and storage services, such as the Android frame buffer and SQLite. The frame buffer replaces the X11 graphics system found in Linux. SQLite acts as the backend for most platform data storage. Windows Phone and iOS have similar backends, called *Isolated Storage* and *Core Data*, respectively.

Applications running on the Android platform can utilize services offered by the application layer to enrich the experience. For example, the *Notification Manager* offers a common mechanism for delivering notifications to the user. The manager can, for example, delay notifications if the user is in a phone call. On

Figure 2.3: Overview of the Android system architecture [64].

the Windows Phone platform, similar functionality is found in the *Notification Service*, and iOS applications can use the *Core Notification Framework*.

Another example is the Android Location Manager, which offers current GPS coordinates to applications that desire information about the geographical location of the mobile device. The purpose of this service is to have a central place to acquire coordinates, so that the interactions with the actual GPS hardware is minimized, and the manager can offer cached coordinates if applications concurrently query for location. Figure 2.4 shows how an Android application utilizes the Location Manager to retrieve updated GPS coordinates, and which components are activated as part of servicing the query. On the Windows Phone platform, the geographical location service is called the *Location API*, and on iOS it is known as the *Core Location Framework*.

2.2.1 Isolated Execution

According to a Nielsen survey [66], an average smartphone user actively used more than 26 applications every month in Q4 2013. With application areas

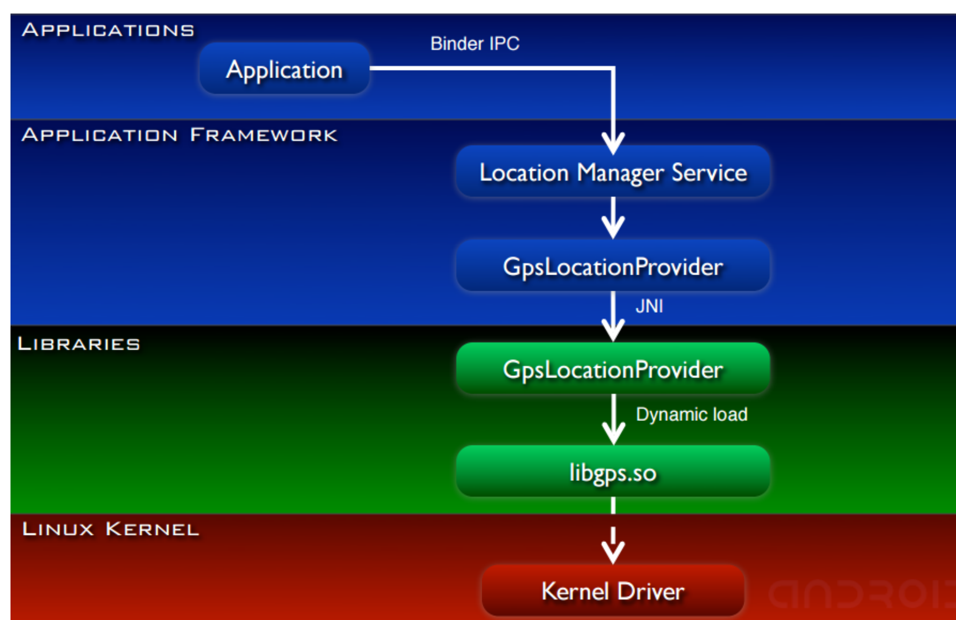


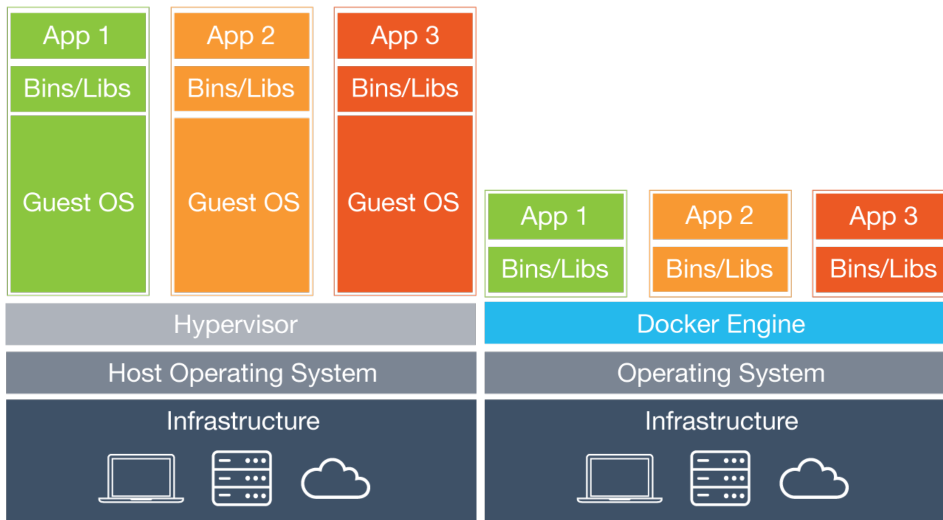
Figure 2.4: How the Location Manager is used on Android [64].

varying from banking and finance, to social media and games, securing the information of each application is essential. A game should not be able to access account information from a bank application, and vice versa.

Isolated execution platforms are used to isolate the execution of application code from the underlying OS and from other applications. They ensure that any failures or anomalies are contained in an isolated environment and do not affect the performance or correctness of the OS or other applications.

An isolated execution platform can be implemented using both software and hardware approaches. Hardware techniques will generally promise the highest level of isolation. Using low level CPU instructions, two-way isolation between executing code and the OS can be achieved, for example with ARM *TrustZone* [67] or, on Intel *x*86-64 CPUs, Software Guard Extensions (SGX) [68]. Several academic works investigate the benefits of hardware techniques for isolation. [69] leverages ARM *TrustZone* to build a trusted runtime that allow applications to execute in an environment isolated from the OS and other applications. Similarly, [70] utilizes Intel SGX to provide a mutually distrusted execution environment for unmodified applications.

The most prevalent software approach to isolation on mobile devices is virtualization [71]. A major advantage with virtualization is that the application developer can disregard the underlying hardware platform, and deploy the



(a) Applications running in a VM. **(b)** Applications running in Docker.

Figure 2.5: Difference between applications running in a VM and Docker [74].

same application on different CPU architectures.

Virtualization can be achieved on several levels of the software stack, ranging from a full fledged OS running on top of a hypervisor, to process virtualization, where an application is hosted in isolation inside another process. Figure 2.5a illustrates a hypervisor, such as VMware [72] or Oracle VirtualBox [73], isolating the execution of an entire guest OS and its applications. In comparison, Figure 2.5b shows how Docker [74], a process virtualization engine, isolates the execution of applications. While the hypervisor approach may offer more complete isolation, it also carries the additional cost of initializing and hosting the virtual guest OS.

2.2.2 Application Runtime

Mobile OSs generally implement an *application runtime* that serves as an isolated execution platform for applications. On Android, a process virtualization technique is used, where applications can be implemented in Java and are executed on top of a Virtual Machine (VM) based on the Java Virtual Machine (JVM) specification. This way, applications are isolated from the underlying Android OS, and need not relate to a variety of concrete machine architectures.

Windows Phone is based on the Windows kernel, and has an application runtime called the Windows Runtime (WinRT), where applications are written in C# or

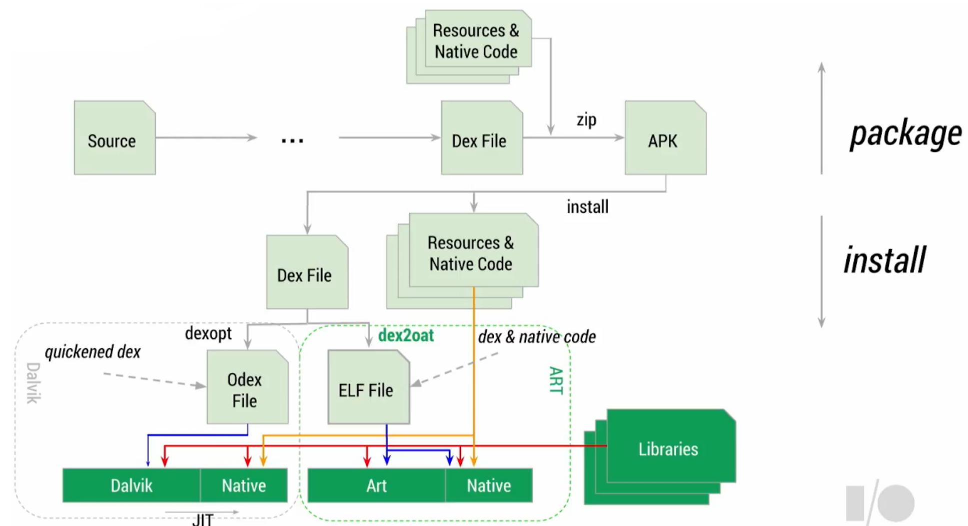


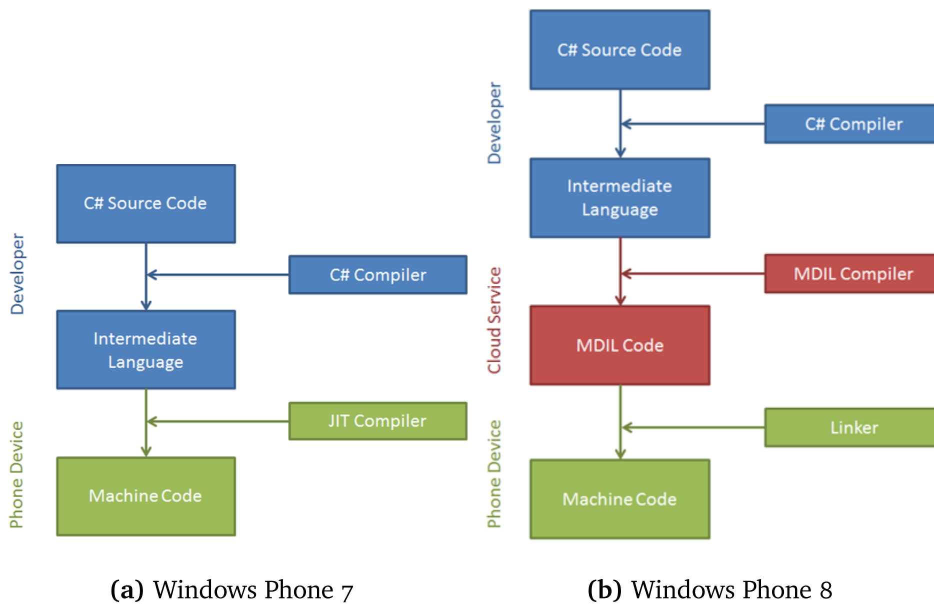
Figure 2.6: Lifecycle of an Android application from source to running, both on Dalvik and ART [76].

another .NET language. Apple's iOS is based on the FreeBSD XNU kernel, and features a proprietary Objective-C runtime system. Applications are written in Objective-C, or in higher-level languages that compile into Objective-C, such as Swift [75].

The first JVM implementation that was used on Android is called Dalvik. Dalvik applications are written in Java, and compiled using a regular Java compiler to bytecode for the JVM. The Java bytecode is then translated to Dalvik bytecode using a tool called *dx*. This tool collects all of the Java class files that comprise an application and converts them into a single Dalvik Executable (DEX) file, which resembles a Java Archive (JAR) file, but conserves space by coalescing duplicate strings and constants.

A DEX file can be bundled along with resource files, such as graphics, and compressed into an Android Application Package (APK) file, which is the package format used to distribute and install application software on Android. Upon installation on the mobile device, Dalvik will further optimize the DEX file by inlining data structures and functions into an optimized version called Optimized Dalvik Executable (ODEX). When executing the application, bytecodes are interpreted, but Dalvik will continuously profile the execution and compile frequently executed segments of bytecode into native machine code using a Just-In-Time (JIT) compilation technique.

On Android 5.0, Dalvik has been replaced in favor of the new Android Runtime (ART), which takes the approach of compiling the entire application to native



(a) Windows Phone 7 **(b)** Windows Phone 8

Figure 2.7: Lifecycle of an Windows Phone application [78].

machine code during installation to the mobile device. While this incurs some additional cost at install time, ART demonstrably improves overall execution efficiency and reduces power consumption [77].

A similar evolution has occurred on the Windows Phone platform. On Windows Phone 7, the source code was compiled into Common Intermediate Language (CIL), which is the intermediate format for all .NET languages, and then converted to binary machine code using JIT compilation every time a method was invoked. However, JIT compilation can add significant overhead as it entails both parsing of metadata, code validation, and compilation into machine code. To make matters worse, the compiled code was discarded when the application terminated, and thus recreated from scratch every time the application executed.

This was changed in Windows Phone 8, by using a cloud service to compile the CIL image of the application into a Machine Dependent Intermediate Language (MDIL) representation before installing the application on the mobile device. MDIL is a format that resembles the final machine code, but has placeholder tokens for platform specific machine code. These placeholders are replaced during installation on the device to optimize for specific hardware.

With several different mobile OSs, all having different runtimes and employing different programming languages, developers face challenges when developing applications that target multiple platforms. Even for simple applications, this

Figure 2.8: The Xamarin platform binds native iOS and Android SDKs to the .NET platform [79].

demands more experience from developers, and can escalate costs as a result of code duplication across languages.

Xamarin [80] is a developer platform that seeks to address this problem by allowing developers to code native, cross-platform Android, iOS and Windows Phone applications in C#. It contains ported versions of .NET for iOS and Android, called *Xamarin.iOS* and *Xamarin.Android*, respectively. These exist to create .NET bindings to the native Software Development Kits (SDKs), so that the features of the underlying OS can be accessed from C#. Based on this, Xamarin builds a stack of abstractions that offers a unified way to create portable applications, as shown in Figure 2.8. This allows developers to create rich user interfaces and use native features like notifications, graphics, and animations from a shared C# code base that works for all platforms.

The richness of interfaces and functionality available to a mobile application simplifies the use of advanced hardware and sensors on the device. But mobile applications often need more resources than the device can provide, and will to varying degrees rely on cloud services to provide additional functionality-enrichment.

2.3 Cloud Services

There exist a plethora of cloud services available for mobile devices. For example, Google Maps can be used in conjunction with a device's GPS service to provide a

real world map with indicators where the device is located, perhaps augmented with some points-of-interests close by. Microsoft offers similar service through Bing Maps, and Apple has Apple Maps.

Other popular cloud services such as Google Gmail provide e-mail, calendars and contacts, and will push out notifications for new e-mails received, schedule reminders for calendar events, and keep contacts in-sync across devices. Microsoft have the same offerings through Outlook and Apple has iCloud.

The mobile device local storage can be expanded by automatically uploading pictures taken with the camera to a cloud storage service. Google Drive, Microsoft OneDrive, Apple iCloud, Dropbox, and MEGA are only a few of the providers that will let you automatically upload photos. All of them offer an initial free capacity ranging from 1 GB to 50 GB of storage, that can be expanded by paying a fee.

There are also cloud picture services that have an additional social aspect. These services allow other users to view, share, and comment on uploaded pictures. Among the most popular offerings we find Instagram, Pinterest, Flickr, and Photobucket.

YouTube has a similar social service for videos. With 65 hours of new videos being uploaded every minute [81], YouTube is the largest video sharing network in the world. Other social networking services can provide features beyond image and video storage and sharing. Twitter, for example, offers sharing of short messages, while LinkedIn offers sharing of work experience and education.

The biggest social network is Facebook, with its 1.3 billion users [82]. Of these, 680 million users access the network from a mobile device. Facebook offers a compound way for users to interact with each other, with both picture sharing, status updates, event and group communication, and a platform for developers to create extensions like games or even stores selling actual products.

These cloud services are architected in different ways, but common building blocks are storage and computation services. Storage services can range from block storage to column and object-oriented databases. Mobile applications use storage services for content, user profiles, authentication, and session state. Furthermore, cloud services that serve content in one way or another will most likely utilize geographically distributed storage, as well as advanced caches and indexes to lower the latency for users requesting content.

Compute resources can be anything from containers hosting scripts, distributed frameworks for analytical computations, to full-fledged VMs. For example,

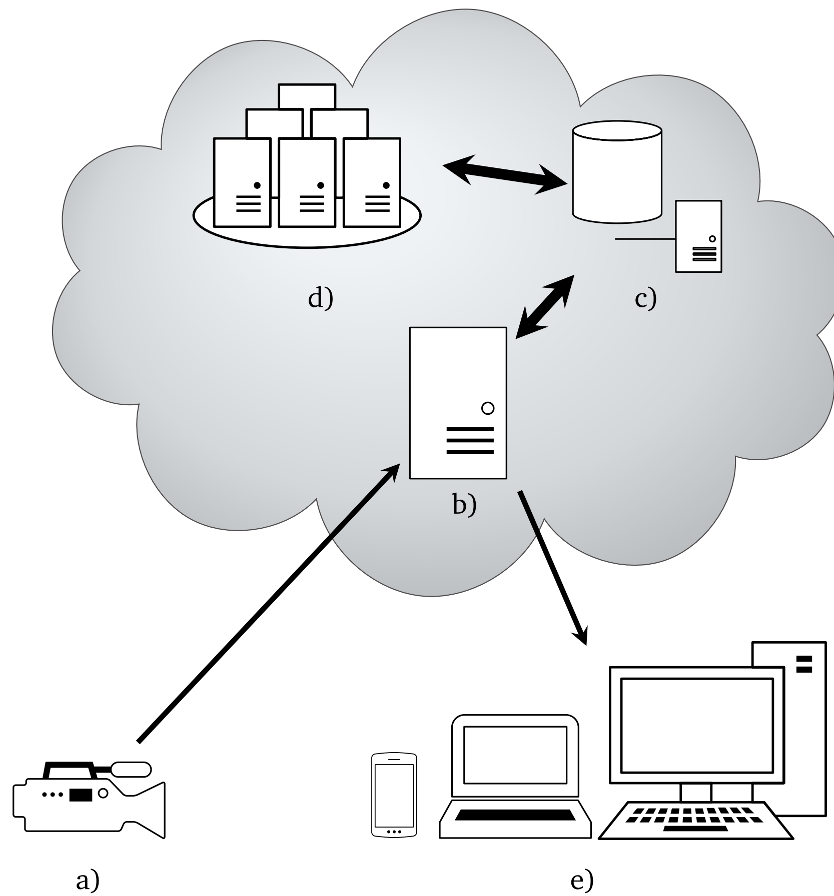


Figure 2.9: Illustration of a generic cloud video sharing service. Videos are uploaded from a recording capable device at *a)* to a cloud front-end at *b)*, before they are stored in a cloud database at *c)*. Compute resources will re-encode the video in various formats at *d)*, before consumers can request a format suitable for the viewing device at *e)*.

compute resources can be used to produce images scaled and optimized for a particular mobile device screen size. In a similar fashion, cloud video services utilize compute resources to scale and re-encode videos to optimize viewing for the requesting device.

Figure 2.9 illustrates how a generic video sharing service could be architected as a cloud service by composing other basic services. Video is recorded and uploaded from a recording device at *a)* to a front-end in the cloud at *b)* which exposes a public Application Programming Interface (API). Internally the cloud service will store the video in a database at *c)*, and instruct compute resources at *d)* to re-encode the video in multiple formats for consumers. Consumers at *e)* can then in turn access the public API and request a version that fits the

display capabilities of the requesting device.

Common for most cloud services is that heavy analytical computations are active under the surface, producing recommendations and deducing trends based on user activity. These computations can span multiple dedicated datacenters, running incremental MapReduce [83, 38, 84] jobs to continuously compute the latest trends and recommendations for the users of the network.

Applications utilizing these services do not have to consider the underlying complexity, and are instead presented with a well formed RESTful [85] API allowing usage from different devices all over the world.

Previously, these cloud services consisted of proprietary code running on VM-based offerings from cloud providers such as Amazon. Designing new cloud services that could compete with existing services was expensive, not only in terms of infrastructure cost, but also in terms of development costs and required experience. To ease development of new cloud services, a class of modular cloud services is emerging that factors out the most common functionality of cloud services into more generic and reusable frameworks.

2.4 Developing Mobile/Cloud Applications

Parse [86], Google Cloud Platform [87], and Amazon Web Services [88] have platforms that offer modular cloud services. These platforms offer a modularized approach to designing mobile applications that depend on cloud services. Functionality provided by the modules include user authentication and session storage, communication and connectivity with other users, and process management and computations.

Depending on the application, user identification can be needed in varying degrees. Either to associate data with a specific user, or to facilitate location of other users in the application. User authentication can either be implicit by unique device identifiers, or explicit with a username and password. User authentication on most modular cloud platforms can use new credentials created in the application, or make use of an existing profile from e.g. a social network through the OAuth [89] protocol. On the Google Cloud Platform this functionality is found in the *users* feature of Google App Engine. Amazon offers similar functionality through their *Amazon Cognito* service, and Parse has the *ParseUser* class.

Knowing how users actually use an application is important to further improve the quality and evolution of an application. And as the content grows beyond

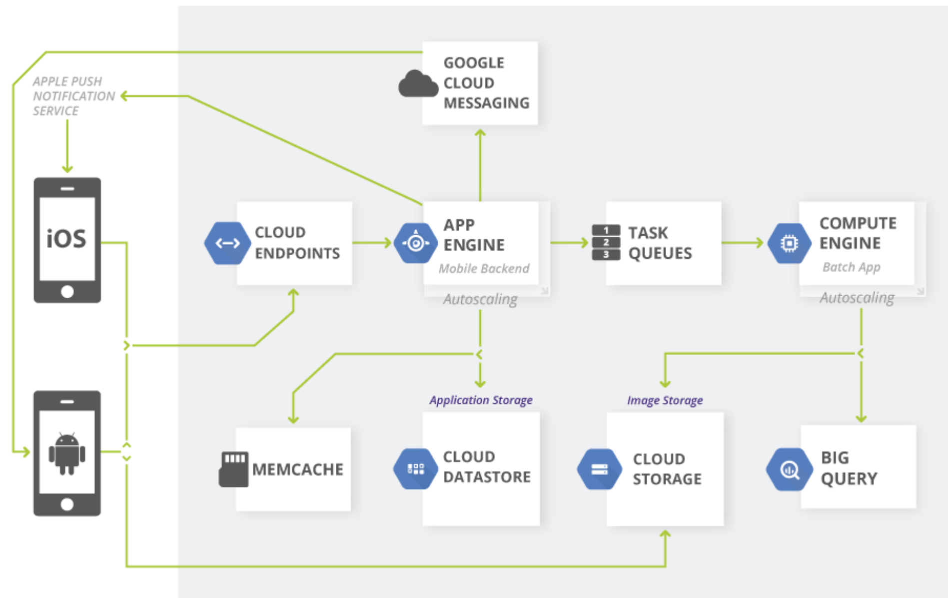


Figure 2.10: Example architecture of a mobile/cloud application composed of cloud modules from the Google Cloud Platform [87].

users' ability to easily digest it, filtering of content based on the user's preferences and previously digested content might be advantageous. These insights can be learned through analytical computations. Analytical computations, both streaming and batch oriented, can be implemented using Google's *Dataflow*, *Amazon Mobile Analytics* or *ParseAnalytics*. These systems allow developers to seamlessly tap into virtually limitless computational power to analyze trends and behaviors, and provide users with content recommendations.

The modular cloud platforms also provide modules for sending notifications to mobile devices, content distribution, and the ability to execute custom code in the cloud. Usually this custom code is expressed in JavaScript using an SDK from the provider, and executed in response to a *GET* request from a mobile device.

Figure 2.10 illustrates an example mobile/cloud application architecture composed of cloud modules from the Google Cloud Platform. The mobile application and the corresponding cloud service are typically developed in tandem. The cloud service executes in the *App Engine* module, and the platform will autogenerate a RESTful API and associated client libraries based on annotations in the service code. The application can in turn utilize the library to access the service through the *Cloud Endpoints*. The App Engine uses autoscaling to handle dynamically changing workloads and *Memcache* provides a shared, in-memory cache to speed up access to recently accessed data. *Cloud Datastore*

provides the application with schemaless object storage with a SQL-like query language, while *Cloud Storage* distributes static content like graphics and media. Notifications can be sent to both Android and IOS devices, using *Google Cloud Messaging* and the *Apple Push Notification service*, respectively.

2.5 Middle Tiers

Mobile/cloud applications have multi-tiered architectures by definition. with a presentation tier running on the mobile device and logic and data storage tiers running in the cloud. Depending on the application, the architecture may have one or more middle tiers positioned on the communication path between the presentation tier and the logic and data storage tier. Figure 2.11 identifies three general areas where a middle tier can be positioned to augment the functionality of the application; a) in the local network, b) between the local network and the cloud, and c) in the cloud.

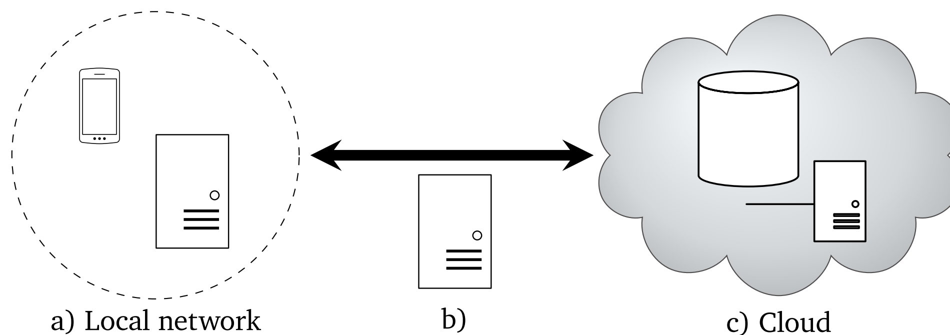


Figure 2.11: Potential locations a middle tier can be positioned to augment functionality of a mobile/cloud application. In the local network close to the mobile device *a)*, on the communication path to the cloud *b)*, or in the cloud *c)* close to the cloud service.

By placing a middle tier in the same local network as the mobile device, the middle tier can communicate with the mobile device with low local network latency. Comet [22] and Cloudlets [21] exploit this low latency communication to increase the available computational resources for the mobile application by offloading the whole or parts of an application to execute in a VM on a local server. In some cases Comet observed a geometric mean speedup of 2.88x for unmodified applications, compared to running directly on the device. MAUI [24] uses a profiler to dynamically offload code to a local MAUI server to save energy consumption on mobile devices.

Between the local network and the cloud, Content Distribution Networks

(CDNs) [26] can be used to distribute static application content from the cloud closer to mobile devices. A CDN consists of three main components: a distribution system, a request routing system and a number of geographically distributed replica servers. The distribution system replicates data marked for distribution from the origin to the globally distributed replica servers. The request routing system redirects requests from clients to a nearby replica to reduce latency.

Having a middle tier in the cloud will increase the communication latency for mobile devices compared to a middle tier situated in the local network, but will significantly reduce latency to cloud services and increase available computational resources. Since the cloud is globally available, availability can also improve when mobile devices roam outside the local network.

The motivation for having a middle tier in the cloud can also be to reduce power consumption on the mobile device, or to increase application performance. CloneCloud [23] is similar to Comet, but uses cloud resources to host the VM instead of local servers. Their experiments show that some applications can achieve as much as a $20x$ execution speed-up and a 20 fold decrease in energy consumption.

Offloading of code from mobile devices to a middle tier can be architected in different ways. Both Comet and CloneCloud are using a profiler to decide which parts of the code to offload. Developing an application with explicit offloading mechanisms can increase opportunities for code offloading, since the developer will have better insight into what the application seeks to accomplish. Sapphire [2] has chosen this approach by designing a distributed programming platform that allows developers to annotate methods in the application that should be offloaded.

As more and more mobile applications offload execution of code to local servers or to the cloud, new challenges arises in determining priorities to make timely responses. These challenges are exacerbated by the varying quality of mobile networks that handle the communication between the devices and the servers. Timecard [14] attempts to address these issues by having a machine learning algorithm adapt its processing time, making a trade-off between response quality and processing time to keep the end-to-end delay for the request below a given threshold.

Offloading code from mobile devices to a middle tier can have several benefits. If the code must access sensitive data, security can be improved by offloading the code and not having the sensitive data stored on local devices. Consistency across multiple devices can be achieved by using a centralized cloud service. Computational resources can be increased by utilizing the elastic nature of

the cloud. Power consumption can be decreased by offloading computational intensive tasks to external servers. And latency can be decreased by having low-latency access to external resources.

However, there are disadvantages by relying on a middle tier as well. Having an application depend on external resources for normal functionality, can restrict the application if the network or cloud provider should be unavailable. Storing data in the cloud can pose privacy issues where computations involve data with confidentiality constraints, or governmental compliance or export restrictions. And depending on external vendors or services can lead to vendor lock-ins, incurring substantial switching costs.

2.6 Summary

This chapter surveyed modern smart phone architectures, and how cloud services are used to enhance the user experience on mobile devices. Applications run in isolation on mobile devices with limited computational resources, and communicate with cloud services that draw upon the elastic nature of the cloud, appearing to have unlimited computational resources. Considering the roaming nature of mobile devices, with slow network links exhibiting high communication latencies, the communication path to the cloud could prove to be the dominant bottleneck. Previous work has investigated how to reduce this latency by utilizing a middle tier component in various configurations. In the coming chapters we present our own efforts to create a middle tier component that reduces communication latency for mobile/cloud applications.

3

Optimizing Reads from the Cloud

Cloud database services are convenient building blocks for mobile/cloud applications. By serving as a highly available and resilient point of contact, a central cloud database service can greatly simplify the architecture. For example, devices can locate peers simply by looking up the relevant membership information in the cloud, and configuration or software updates can be retrieved from a central location. A generic cloud database service can cover these needs while guaranteeing both availability and scalability, alleviating any concerns that the central component should become a bottleneck.

A downside of relying on a cloud database service is that all devices must communicate directly with the central service, generating load that translates into increased operational costs. A natural step to reduce both latency, load, and the associated charges from the cloud service provider would be to employ caching of data outside the cloud, closer to the client devices. However, this introduces new architectural complexity, in the form of a separate caching infrastructure that must be deployed as a middle tier between the devices and the cloud.

Our overall goal is to build a generic middle tier that *exploits existing infrastructure* to reduce latency for mobile/cloud applications. Given the popularity of cloud database services, and the apparent opportunities for reducing latency

through caching of database values, we initially focused on this use case.

It is common practice for cloud services to employ caching systems such as Memcached [90] and Redis [91] to alleviate database load. These systems are effective in reducing service latency and financial costs, but operate within the cloud infrastructure. Content Distribution Networks (CDNs) employ caching facilities in close proximity to clients [92, 93, 94, 95, 96]. These networks reduce the demand on centralized servers for distribution of web and, in particular, multimedia content. The networks operate by routing a client request to a server holding a copy of the requested content. The content has often been pushed in advance to the replica server by the provider, to prepare for expected demand. This pre-copy approach is not suitable for the dynamic nature of database content.

First we considered the optimal placement of a middle tier that performs caching. Looking at the architecture of a mobile network in Figure 2.1, we identified three potential locations: Close to the mobile device, in the Mobile Network Operator (MNO) infrastructure, and between the MNO and the cloud. Placing the middle tier close to the mobile device would provide very low latency lookups, but fewer devices would be able to share the same cache. Conversely, placing it between the MNO and the cloud would allow more devices to share the same cache, perhaps leading to a higher hit rate, but potential latency savings for individual cache hits would decrease.

Placing the middle tier in the MNO infrastructure would provide a good compromise, with many devices sharing the same cache while still maintaining low latency access. Deploying new components in the MNO infrastructure would be both costly and practically infeasible. However, when considering the range of existing infrastructure, we realized that the *Domain Name System (DNS)* was a potential starting point, since it is globally available for all devices connected to the Internet, and essentially implements a distributed cache. We therefore decided to investigate if DNS could serve as the foundation for a generic database caching layer that meets our requirements.

This chapter presents the result: the Jovaku system [58]. By relaying database operations through the DNS protocol, Jovaku allows results to be cached temporarily in existing servers close to the clients, with corresponding savings in latency and reductions in load on the central database service. Since the DNS service is ubiquitous, and in practice considered to be an integral part of being connected to the Internet, the availability of this caching layer is unparalleled. Our approach is generic and reusable across both database services and applications, and does not require any special devices capabilities beyond a standard network stack.

In the rest of this chapter we first outline the basic features of DNS, and elaborate on why we chose to build on DNS for Jovaku. Then we detail the overall architecture of Jovaku and the implementation.

3.1 The Domain Name System

DNS is a hierarchical naming system [97] designed for high availability, comprising a worldwide network of name servers that communicate with clients and with each other using the DNS protocol. Its most common use is to associate human-readable *domain names*, such as *google.com*, with numeric IP addresses such as `109.105.109.219`.

The domain namespace is partitioned into *zones*, for which administrative responsibility has been delegated to separate managers. A zone has one or more *authoritative* servers responsible for replying to DNS queries pertaining to the zone. Queries are usually lookups for specific *labels*, which are associated with *Resouce Records (RRs)* of various types. For example, an IP address is of type A for IPv4 and AAAA for IPv6, an MX record identifies a mail server, and a geographical location can be stored in a LOC record. Arbitrary text values can be stored using TXT records. A single label can also group multiple RRs into a *Record Set (RS)*.

Although not intended to be a general purpose database, DNS is used for more than just resolving Internet names. Spotify, for example, uses DNS both to locate access points, to prioritize servers, and to store Distributed Hash Table (DHT) configuration for thousands of servers, serving tens of millions of users [98]. Another example of innovative DNS applications is David Leadbeater's Wikipedia over DNS service [99]. With the service, clients can issue DNS queries for Wikipedia articles and receive TXT replies with an excerpt of the contents.

DNS also supports updates, although these tend to be rare in the context of mapping domain names to IP addresses. Updates can overwrite old records or append new records to record sets. Transactions can be used to perform multiple updates atomically, subject to specified preconditions being satisfied.

Caching plays a central role in DNS. Every record has several associated Time To Live (TTL) values that specify how long the record can be cached at various levels of the system. Updates must be sent to an authoritative server for the zone. Lookups, on the other hand, can be served by any server that has the relevant records in its cache, subject to TTL constraints. If a server receives a query that cannot be served from its cache, it will forward the request to one

of the authoritative servers for the zone. There is no cache coherency protocol; instead, the TTL values determine how long it may potentially take for an updated value to become visible to all clients.

The appeal of using DNS in our research stems from its maturity. DNS is already deployed and proven to work on a global scale, with extremely high availability. By experimenting with DNS, we can thus obtain truly realistic results for a globally distributed system, for example with regards to latency measurements. The basic functionality of DNS corresponds well with the features found in a NoSQL database, and at its core, DNS is a distributed cache. This makes our overall approach feasible.

Having a weak consistency model, DNS imposes certain limitations. For instance, applications may want to control the contents of their cache by invalidating cached values. This is not directly supported by the DNS protocol, which relies on the simple TTL system. However, [100] proposes a proactive DNS cache update protocol to provide strong consistency for cached DNS values.

DNS imposes no access restrictions for lookups, so anyone can read data that is cached in DNS, provided they know or can guess the relevant labels. Sensitive data must therefore be encrypted by applications before storing it in the database, although this may arguably be a sensible precaution anyway.

With Jovaku, there is the concern of disrupting regular name resolution traffic with application-specific database caching that should have been deemed less important. Still, the load imposed on DNS by our approach needs not be excessive, and from an overarching point of view, caching can serve to reduce the total load on the network as a whole. Another option is to use one DNS server for regular domain name resolution while another DNS server, such as Google's public DNS server, is used for database caching.

In summary, this chapter shows how certain applications can make significant performance gains by leveraging the freely available DNS infrastructure. They are relevant and potentially useful both for improving current DNS infrastructure, and for building middle tier caching services that are deployed independently of DNS.

3.2 Jovaku Architecture

Jovaku relays database operations through the DNS protocol, allowing results to be cached by DNS servers close to the clients. The key space of the database

is mirrored as labels in a specific domain, whose authoritative name server runs in the cloud, acting as a *relay-node*. This exposes the database for reading by any and all DNS resolvers, such as the common *dig* utility. For example, if a database mirrors its key space as labels in the *jovaku.com* domain, resolving the label *x.jovaku.com* will yield the database value associated with the key *x*.

Figure 3.1 shows how an application will access a cloud database with and without Jovaku. The baseline behavior is to use an Application Programming Interface (API) to directly access the database service, as in Figure 3.1a. These requests are usually sent over HTTP, and will have a constant cost equal to the round-trip time of the communication path to the cloud that hosts the database.

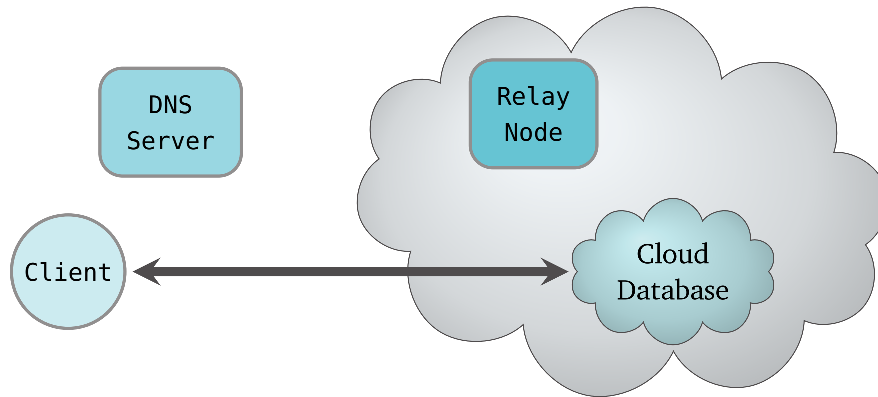
With Jovaku, requests are sent over the DNS protocol to the local DNS server, which will forward lookups to the relay-node if they cannot be served from its local cache. Lookups that miss the cache thus take a slight detour, as illustrated in Figure 3.1b. To minimize this detour, the relay-node should be deployed in the cloud, close to the database service. On the other hand, lookups that hit the cache will only make a short round-trip to the local DNS server, as in Figure 3.1c, with a corresponding improvement in latency.

Updates can either be submitted over the DNS protocol, through the relay-node, or use the API and go directly to the cloud database over HTTP. Barring routing artifacts that favor one protocol over the other, going through DNS will generally not improve latency for updates, since there is no potential for a cache hit. The required access credentials will differ, however. Performing an update through DNS requires a transaction signature (TSIG) that matches keys stored at the authoritative name server, while the database API will generally implement a separate access control mechanism.

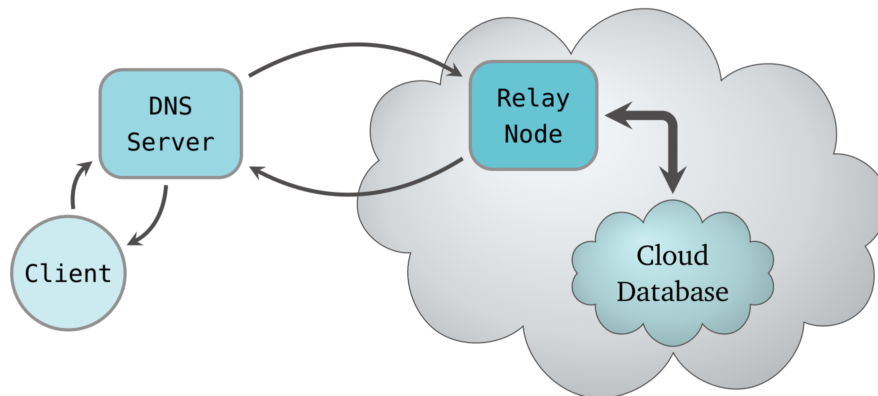
As detailed in the next section, our current implementation of this architecture targets the Amazon DynamoDB [101] cloud database service. However, the general access pattern is not specific to DynamoDB, and should be the same for any other underlying database service.

3.3 The Relay-Node

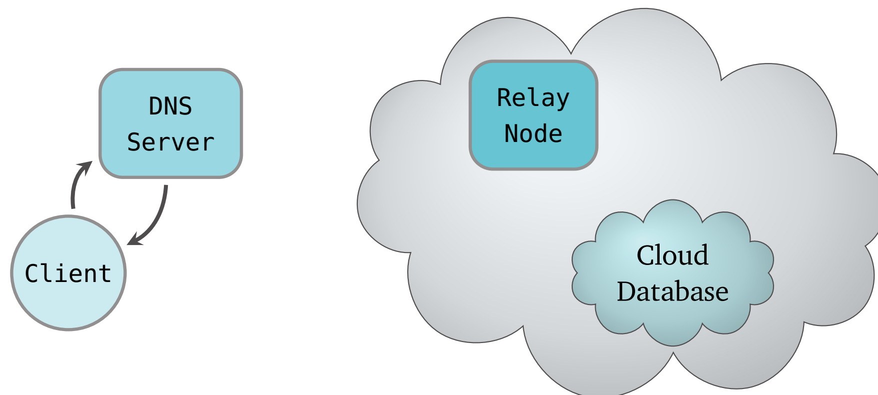
Our implementation targets Amazon DynamoDB, a popular NoSQL cloud database service, and can potentially be adapted to work with any similar services. To obtain the lowest possible latency for database operations, we provisioned an Amazon EC2 instance in the same availability zone as the



(a) Baseline (without Jovaku).



(b) Cache miss with Jovaku.



(c) Cache hit with Jovaku.

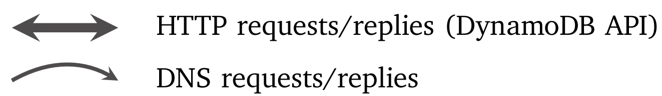


Figure 3.1: Database lookups with and without Jovaku.

DynamoDB service. For our authoritative name server we had the choice between implementing a new server from scratch or modifying an existing server to suit our needs.

Since we seek to exploit existing infrastructure in this dissertation, we chose the latter, and installed the standard *Berkeley Internet Name Domain (BIND)* [102] server. BIND is the most widely used name server, and supports Dynamically Loadable Zone (DLZ) drivers [103]. DLZ drivers are customized extensions for BIND written in C, that implements specialized resolution of DNS queries for a zone. The idea was to implement a DLZ driver that handles all interaction with DynamoDB, and avoid touching the stable code base that implements core BIND functionality. An overview of the composition of the resulting relay-node can be seen in Figure 3.2.

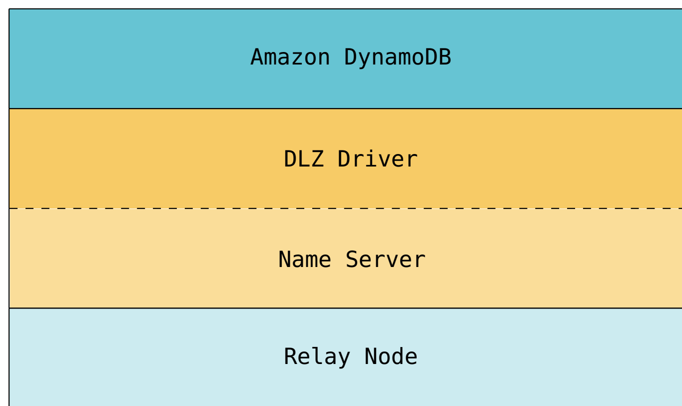


Figure 3.2: Overview of the Jovaku architecture.

While Amazon provides official Software Development Kits (SDKs) for the DynamoDB service in several programming languages, there is no C version of this SDK. So to enable our DLZ driver to access DynamoDB, we also implemented a C version of the DynamoDB SDK, according to Amazon's specification [104]. Our implementation uses the *curl* [105] library to drive HTTP traffic, the *jansson* [106] library for formatting and parsing requests and responses in JavaScript Object Notation (JSON) format, and the *BeeCrypt* [107] library for generating cryptographic signatures. The resulting DLZ driver with the required Amazon DynamoDB operations encompasses 1439 source lines of code.[1]

We have used TCP tracing to verify externally that our implementation behaves similarly to Amazon's official .NET SDK, and has identical performance. For both implementations, the time to execute a request is dominated by the latency of the HTTP request. The time needed to construct the request and parse its

1. Determined using David A. Wheeler's 'SLOCCount'.

reply is insignificant. In a trial with 1000 requests, both implementations performed withing ±3 ms of the latency measured from the test machine to the Amazon DynamoDB node. Both implementations reuse the same connection for consecutive HTTP requests.

Code Listing 3.1: The interface that needs to be implemented to create a DLZ driver.

```
// Initialization and destruction of driver
isc_result_t dlz_create(const char *dlzname, unsigned int argc,
    char *argv[], void **dbdata, ...);
void dlz_destroy(void *dbdata);

// Verify driver belongs to zone
isc_result_t dlz_findzonedb(void *dbdata, const char *name);

// Lookups
isc_result_t dlz_lookup(const char *zone, const char *name,
    void *dbdata);

// Zone transfer
isc_result_t dlz_allowzonexfr(void *dbdata, const char *name,
    const char *client);
isc_result_t dlz_allnodes(const char *zone, void *dbdata,
    dns_sdlzallnodes_t *allnodes);

// Dynamic DNS Updates
isc_result_t dlz_configure(dns_view_t *view, void *dbdata);

isc_boolean_t dlz_ssumatch(const char *signer, const char *name,
    const char *tcpaddr, const char *type, const char *key,
    uint32_t keydatalen, uint8_t *keydata, void *dbdata);

isc_result_t dlz_newversion(const char *zone, void *dbdata,
    void **versionp);
void dlz_closeversion(const char *zone, isc_boolean_t commit,
    void *dbdata, void **versionp);

isc_result_t dlz_addrdataset(const char *name, const char *rdatastr,
    void *dbdata, void *version);
isc_result_t dlz_subrdataset(const char *name, const char *rdatastr,
    void *dbdata, void *version);
isc_result_t dlz_delrdataset(const char *name, const char *type,
    void *dbdata, void *version);
```

DLZ drivers are compiled into dynamically linked libraries that are loaded at runtime into the name server. BIND specifies a DLZ interface that drivers must implement; this interface is listed in Code Listing 3.1. When a zone is configured to use a DLZ driver, queries for that zone are directed to the driver, through the DLZ interface. To configure BIND to use our DLZ implementation

for the *jovaku.com* domain, using our DynamoDB table named “dns”, we enter the following in *named.conf*:

```
dlz “jovaku.com” {
   database “dlopen dlz_dynamodb.so jovaku.com dns”;
};
```

This will instruct BIND to load our DLZ implementation and invoke the `dlz_create` function with “jovaku.com” and “dns” in the argument list. Our implementation of `dlz_create` will initialize curl, set up necessary data structures and send an initial query to Amazon DynamoDB requesting the Start of Authority (SOA) record and the associated Name Server (NS) records from the “dns” table. These records must be supplied with all replies, so caching them at startup will improve performance. Table 3.1 shows the layout of the DynamoDB table. The SOA and NS records have an @ prefix to distinguish zone authority from regular labels.

After the DLZ driver has been initialized, BIND will only invoke it for functionality that relates directly to the resolution of labels in the zone specified. The driver does not concern itself with the management of client connections, decoding of requests, or formatting of responses. To help the DLZ driver communicate responses back to BIND without relating to the specifics of the DNS protocol, BIND provides a set of callback functions—as seen in Code Listing 3.2—that should be used to pass results back.

Code Listing 3.2: Callbacks provided by BIND, for communicating results from the DLZ driver back to BIND.

```
typedef void log_t(int level, const char *fmt, ...);

typedef isc_result_t dns_sdlz_putrr_t(dns_sdlzlookup_t *lookup,
    const char *type, dns_ttl_t ttl, const char *data);

typedef isc_result_t dns_sdlz_putnamedrr_t(
    dns_sdlzallnodes_t *allnodes, const char *name,
    const char *type, dns_ttl_t ttl, const char *data);

typedef isc_result_t dns_dlz_writeablezone_t(dns_view_t *view,
    const char *zone_name);
```

When receiving a DNS query, BIND will invoke the `dlz_lookup` function with the label being queried along with the zone. The DLZ driver will then issue an Amazon DynamoDB query with two key conditions: (1) the label is concatenated with the zone to create a hash key that will match the *Label*

Table 3.1: Layout of the DynamoDB table.

Label	Type	Data	TTL
@.jovaku.com.	SOA	ns1.jovaku.com. robert.cs.uit.no. 1337 3600 30 3600 10	3600
@.jovaku.com.	NS	{"ns1.jovaku.com.", "ns2.jovaku.com."}	3600
ns1.jovaku.com.	A	{"54.154.89.61"}	3600
ns2.jovaku.com.	A	{"129.242.19.153"}	3600
x.jovaku.com.	TXT	{"Some example data"}	10
y.jovaku.com.	TXT	{"This data will not be cached"}	0
z.jovaku.com.	TXT	{"Data item1", "Data item2"}	60

column in our database and (2) a range query over all possible types from the *Type* column. The results are passed to BIND using the `dns_sdlz_putrr_t` callback. The cached authority section retrieved when initializing the DLZ driver is also passed back to BIND.

The two functions `dlz_allowzonexfr` and `dlz_allnodez` are related to zone transfer. This functionality has traditionally been used by slave name servers to obtain an updated copy of the zone file. This has also been used by some domain registrars that require correctness in the zone information to uphold pointers to the authoritative name servers [108]. `dlz_allowzonexfr` returns success for zones that are managed by the driver to indicate that zone transfers are allowed. `dlz_allnodez` will retrieve all entries belonging to the given zone, and pass them to BIND for transfer.

The remaining functionality of the DLZ driver is related to *Dynamic DNS*, which enables updates to DNS data. The `dlz_configure` function simply queries the DLZ driver whether dynamic DNS is supported or not. `dlz_ssumatch` will present the DLZ driver with both the key and the TCP source address of the entity that attempts to start a new update transaction, so that authentication can be performed.

`dlz_newversion` and `dlz_closeversion` mark the start and end of a transaction, respectively. Once a new transaction has been started—and authentication has succeeded—updates to the DNS zone data can be issued through one of

the following functions:

- `dlz_addrdataset` will either add a new RR to an existing RS, or create a new RS with one RR.
- `dlz_subrdataset` will either remove a RR from a RS if there are multiple RR or remove the entire RS if if contains only one RR.
- `dlz_delrdataset` will remove the entire RS regardless of how many RRs it contains.

A DNS update transaction can be performed using the common *nsupdate* tool. DNS transactions can contain zero or more preconditions, add commands, and delete commands. Code Listing 3.3 shows an example transaction performed with nsupdate on the *jovaku.com* domain. The precondition at line 3 states that the TXT label *key* in the *jovaku.com* domain should be absent. Upon receiving this transaction, BIND will perform a normal lookup of that label, invoking `dlz_lookup` in our DLZ driver. If BIND gets a non-empty answer, the transaction will terminate. Otherwise, `dlz_ssumatch` will be invoked, instrumenting the driver to perform access control for the label using the TSIG key for the zone provided at line 2. If the key matches `dlz_newversion` will be invoked, signifying that updates will follow. In the example `dlz_addrdataset` will be called, instructing the driver to create a new entry in the database with the label *key*, a TTL of 40 seconds, and "Some Data" as the value. After the call has completed `dlz_closeversion` will be invoked, indicating the end of the transaction.

Code Listing 3.3: Example DNS update transaction performed with nsupdate on the *jovaku.com* domain.

```
1  # nsupdate
2  > key jovaku.com TSIG
3  > prereq nxdomain key.\phdSmallname{}.com TXT
4  > update add key.\phdSmallname{}.com 40 TXT ‘‘Some Data’’
5  >
```

DynamoDB Interface

Before queries can be sent from the DLZ driver to Amazon DynamoDB, a JSON-formatted message must be constructed according to the specifications in the documentation [104]. The query for the SOA and NS records sent at startup can be seen in Code Listing 3.4. The message contains two directives: (1) the "TableName" property specifies that the query should be performed in the "dns"

Code Listing 3.4: The body of the JSON request for the SOA and NS records.

```
{
  "TableName": "dns",
  "KeyConditions": {
    "Label": {
      "AttributeValueList": [
        {
          "S": "@.jovaku.com."
        }
      ],
      "ComparisonOperator": "EQ"
    },
    "Type": {
      "AttributeValueList": [
        {
          "S": "SOA"
        },
        {
          "S": "NS"
        }
      ],
      "ComparisonOperator": "BETWEEN"
    }
  }
}
```

table and (2) the "KeyConditions" property specifies two conditions: the label must be equal to "@.jovaku.com" and the type must be in the range between SOA and NS.

After the JSON message has been created, the HTTP header—as seen in Code Listing 3.5—must be constructed. Other than the standard directives for host, content-type and length, Amazon specifies a target directive—*x-amz-target*—where the API version and query type must be entered. The *Authorization* directive contains a signature that is created with a keyed hash function using a private key associated with the database, hashing the JSON message and the HTTP headers using the private key associated with the DynamoDB service.

Code Listing 3.5: The header of an HTTP *query* request to Amazon DynamoDB. The Authorization directive has been truncated for readability.

```
POST / HTTP/1.1
host: dynamodb.eu-west-1.amazonaws.com
x-amz-date: 20150704T102030Z
x-amz-target: DynamoDB_20120810.Query
Authorization: AWS4-HMAC-SHA256 ...Signature=145b1567ab3c5...
content-type: application/x-amz-json-1.0
content-length: ##
connection: Keep-Alive
```

Updates are performed similarly, but the *x-amz-target* directive in the HTTP header is changed to *UpdateItem*. The JSON payload also has a slightly different layout, and an example update request can be seen in Code Listing 3.6. The update request will replace the value of the *x.jovaku.com* TXT label with "Updated Data Value".

3.4 Client Library

When Jovaku is deployed as a cache for a database, lookups in the database can be made using standard name resolution utilities such as *dig*. However, we also provide a client library for convenient programmatic access, without manually constructing DNS requests and parsing replies. This library mimics the interface of a standard NoSQL database, and currently comes in two flavors: a Python version, which we use for development and performance testing on Linux, and a C# version for Windows and Windows Phone, which we use to develop mobile applications. The Python version also works on Windows, but relies on spawning *nslookup* for DNS lookups, which is inefficient on Windows, due to the higher overhead for process creation. The C# version of the client library encompasses 581 source lines of code, and has no dependencies beyond the standard .NET libraries.

The basic interface of the client library is similar across both languages, and the C# API is shown in Code Listing 3.7. The library is initialized using the `JovakuFactory.Init` method with the zone where the database key space is mirrored, the dynamic DNS TSIG key used for updates, and optionally a list of hostnames for DNS servers to use for lookups. If the list of DNS servers is

Code Listing 3.6: The body of the JSON request for updating the value of the *x.jovaku.com* TXT label to “Updated Data Value”.

```
{
  "TableName": "dns",
  "Key": {
    "Label": {
      "S": "x.jovaku.com"
    },
    "Type": {
      "S": "TXT"
    }
  },
  "AttributeUpdates": {
    "Data": {
      "Action": "PUT",
      "Value": {
        "S": "Updated Data Value"
      }
    }
  }
}
```

omitted, the system’s default DNS configuration is detected and used. When the library has been initialized, the database can be accessed through the `IJovaku` interface:

- Database lookups are performed with the `Lookup` method with the key to look up as the argument. Keys may have multiple associated values, so the method returns a list of the values associated with the key, along with their age, which indicates how long they have been kept in the cache.

- New values can be added to keys using the `Add` method, with a specified TTL, indicating how long they can be kept in the cache after a lookup. If the key does not exist, it will be created.

- Individual values can be removed from a key using the `Remove` method. If the value that is being removed is the last value associated with the key, the key is removed from the database.

Code Listing 3.7: The C# version of the Jovaku programming API.

```
public interface IJovaku
{
    Tuple<List<string>, int> Lookup(string key);

    void Add(string key, string value, int ttl);

    void Remove(string key, string value);

    void Delete(string key);

    Tuple<List<string>, int> Refresh(string key);
}

public class JovakuFactory
{
    public static IJovaku Init(string zone, string ddns_key,
        List<HostName> dnsServers = null) { ... }
}
```

- Alternatively, a key can be removed using the `Delete` method, which also removes all values associated with the key.

- Finally, keys can be refreshed using the `Refresh` method, forcing a fresh set of associated values to be retrieved from the database service, effectively bypassing the local DNS cache.

As noted in Section 3.1, DNS has no built-in mechanisms for purging cached values. Once a label is cached in a DNS server, the server will not forward lookups for that label until its TTL expires. Our implementation of the `Refresh` method works around this limitation with the cooperation of the relay-node. To refresh a key, a special prefix is prepended to the key, and a lookup is performed on that modified key. That key is never cached by the local DNS server, so its lookup is forwarded to the relay-node, which is the authoritative name server. The relay-node recognizes and strips away the special key prefix, performs the database lookup as usual, and returns the result. This takes advantage of a particular DNS feature that allows replies to include records for other labels than those that were explicitly requested. Effectively, the library tricks the local DNS server into forwarding the lookup by adding the special prefix, and this allows the cache to be updated even if the TTL has not yet expired.

Another added feature that Jovaku provides is the calculation of ages. Every value returned as part of a DNS reply has a TTL value, which indicates how long

it can be cached. But there is no information about how long a value has been cached, i.e. how recently the value was retrieved from the authoritative server. Jovaku implements this by automatically encoding the original TTL value as a prefix of all values. (Note that this is a value prefix, not a key prefix, as used in the `Refresh` implementation.) These original TTL values are extracted by the client library and used to calculate the age of each value using simple subtraction. For example, if a value has 100 seconds as its original TTL value, and the DNS reply indicates that it can be cached for 40 seconds, it can be concluded that its age is approximately 60 seconds. Through the combination of ages associated with values and the ability to refresh values as desired, Jovaku applications are given more control over how the local DNS cache is utilized.

Beyond the extraction of original TTL values, as described above, our client library does not interpret values in any way. They are treated as opaque byte strings. Features such as encryption, compression, and object serialization are easily and more appropriately handled by a surrounding layer of abstraction. The role of Jovaku is to optimize access to the database by serving as an unobtrusive caching layer.

3.5 Summary

Jovaku implements a database caching layer situated as a middle tier between the cloud and its clients. By building on DNS, Jovaku taps into a global infrastructure with high availability and excellent geographical coverage. The key space of the cloud database is mirrored as DNS labels that can be resolved in the usual way to retrieve database values. A relay-node in the cloud translates between the DNS protocol and the database API.

This approach has potential to reduce latency when mobile/cloud applications issue read operations to the cloud, and is based on existing infrastructure. While the DNS protocol has support for updates, passing write operations through DNS will not reduce latency, since updates communicate directly to the authoritative server in the cloud. In the next chapter we will explore ways to reduce latency in more general scenarios that may include write operations.

4

Optimizing Writes to the Cloud

Use of cloud-provided services is integral to the operation of mobile/cloud applications. In particular, cloud databases simplify application logic by serving as highly available repositories for critical state. For improved scalability and availability, these databases are commonly NoSQL, with limited support for tabular relations and transactions and with a more relaxed consistency model than a conventional relational database. Queries are issued through a programmatic interface, rather than a domain-specific, high-level query language.

This promotes a usage pattern where multiple, consecutively-issued queries implement a single logical transaction. For example, an atomic update can be implemented as a read of the old value, followed by a conditional write of the new value, with the predicate that the old value remains unchanged. Or a collection of related records can be retrieved in multiple steps, by manually following foreign key references, rather than using higher-level features like joins and subqueries.

When the database is hosted in the cloud, issuing a sequence of dependent queries entails multiple round-trips of communication, and network latency becomes an important concern. For example, as detailed in Chapter 6, we have measured a latency of 50 ms to 350 ms for accessing a specific Amazon

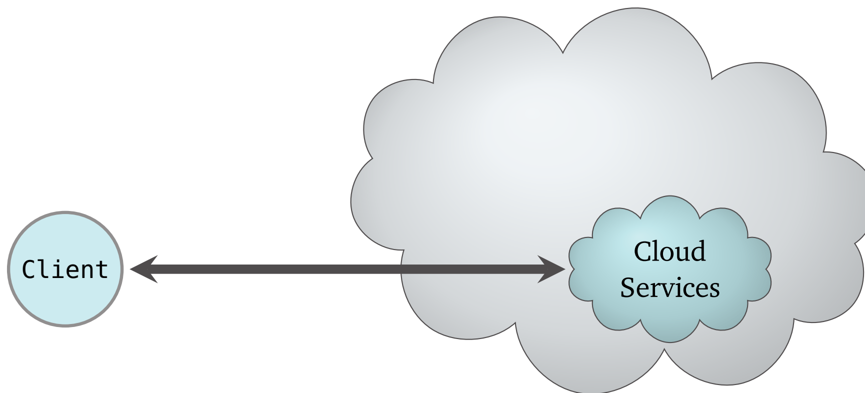
DynamoDB instance from various geographical locations. A study covering 260 global vantage points reports an average round-trip time (RTT) of 74 ms for accessing Amazon EC2 instances [109]. A sequence of queries issued to the cloud can thus result in unwanted delays that are perceptible by users.

Caching infrastructure as outlined in Chapter 3 can in some cases help for a sequence of dependent queries, if some of the keys involved in the queries are cached. But cache misses will inevitably occur, adding an extra round-trip of communication to the cloud database for each miss. One way to alleviate this problem is to move the execution of queries to a middle tier, which is closer to the cloud database. If the entire sequence of queries can be moved as a unit, this can eliminate many round trips between the client and the cloud, substituting them with shorter round trips between the middle tier and the database.

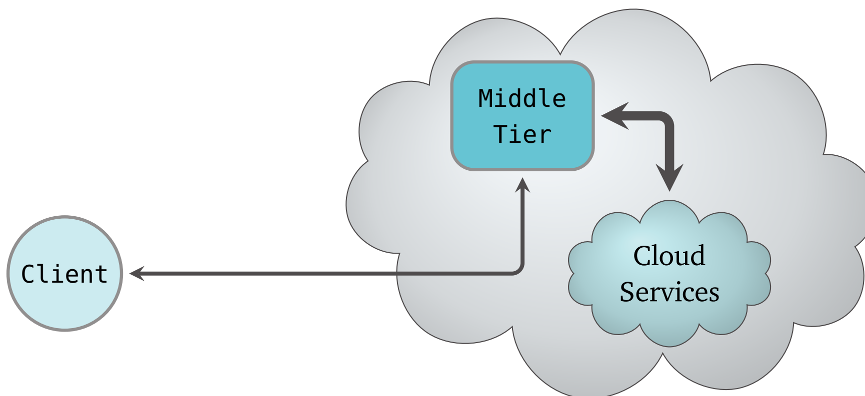
This idea can be generalized to reduce latency for any cloud service where a number of dependent requests are issued sequentially. By moving the code that accesses the cloud service to a middle tier, positioned in close proximity to the cloud service, the code can execute in an environment with lower latency. We refer to this concept as *satellite execution*, and illustrate it in Figure 4.1. Figure 4.1a shows the baseline scenario, where a client must send multiple requests over a high latency mobile network to the cloud to fulfill a task. These can be replaced with a single round-trip as in Figure 4.1b, where code is moved to the middle tier before multiple requests with intracloud latency are issued to the cloud service.

This chapter describes our implementation of satellite execution as an extension of Jovaku. Our approach is to provide a general programming abstraction—*mobile functions*—for location-independent code, which has the potential to either execute locally on a device, or be offloaded to the cloud. As in Chapter 3, we focus on database cloud services as a use case for prototyping and evaluation. Using mobile functions, an application that experiences high latency, or needs to issue a long sequence of database queries, can offload the latency-sensitive code to the cloud and execute it in close proximity to the database service. This ensures low-latency database access on demand, while preserving the programmatic style of database access.

In the rest of this chapter we first outline the extensions to the existing Jovaku architecture. Then we detail the middle tier implementation and how the client library was extended to accommodate the middle tier changes.



(a) Baseline, client communicating directly with cloud services.



(b) With satellite execution.

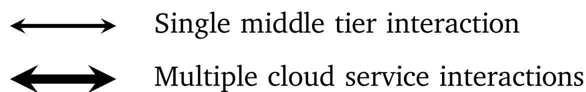


Figure 4.1: How *satellite execution* is applied to eliminate extraneous round-trips of communication between a client and the cloud—by moving code to a middle tier close to cloud services—potentially reducing overall latency.

4.1 Extended Architecture

As described in Chapter 3, Jovaku employs a cloud-side relay-node to bridge the Domain Name System (DNS) infrastructure with Amazon's DynamoDB, translating DNS requests into database queries. The relay-node was placed in close proximity to DynamoDB, in the same availability zone, to avoid extra latency when performing the translations. Since we have similar requirements for the middle tier in our satellite execution concept, integrating this functionality into Jovaku was an intuitive solution. To realize satellite execution,

which hinges on offloading latency-sensitive code, we extended the relay-node with capabilities for hosting and safely executing .NET code. We identified two main components as necessities for this extension:

An execution environment capable of hosting multiple securely isolated *sandboxes* [110]. Each sandbox must be capable of loading and executing code on behalf of its clients, without interfering with other sandboxes or compromising the integrity of the surrounding execution environment. Sandboxing can also be important when multiple applications offload code to the same relay-node, since users can assign different levels of trust to different applications.

A message processor that will receive and process messages sent from clients, and demultiplex and pass those messages to the execution environment. There are several feasible approaches to implementing an efficient message processor. In addition to latency, throughput will be an important concern for the message processor, to minimize the cost of operating the cloud-side relay-node.

An overview of the extended architecture with the new components can be seen in Figure 4.2.

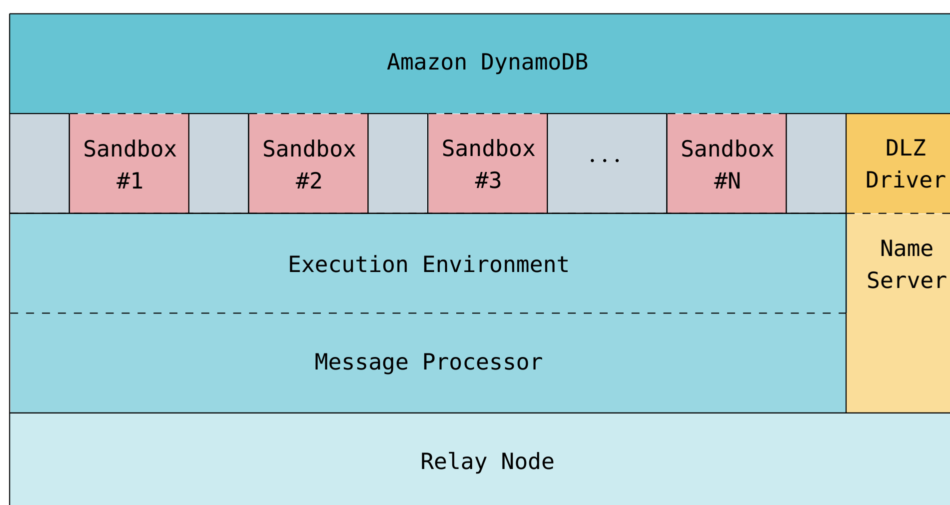


Figure 4.2: An overview of the extended Jovaku architecture where the message processor and execution environment have been integrated into the relay-node. The execution environment is capable of hosting several sandboxes for executing offloaded client code.

To specify offloadable code, we defined the `IMobileFunction` interface seen in Code Listing 4.1. Implementations of this interface are called *mobile functions*,

Code Listing 4.1: Interface that must be implemented by mobile functions.

```
public interface IMobileFunction
{
    Task Execute(IContext ctx);
}
```

as they can be serialized and moved for remote execution on a relay-node. This form of code mobility is classified as weak mobility [111]. The entry point of a mobile function is its `Execute` method, which may be invoked using .NET's task-based asynchronous pattern.

Mobile functions contain user-defined code, and are black boxes to Jovaku. Being implemented in a .NET language, like C#, they enjoy the expressive power of a general-purpose programming language. However, this power must be checked in order to provide a reasonable balance between flexibility and security. Jovaku only invokes mobile functions from sandboxes that are intended to isolate the environment from unwanted side effects, restricting the mobile function's capabilities for actions like file and network I/O. To compensate, Jovaku will let mobile functions access *safe* implementations of selected operations through the `IContext` interface, shown in Code Listing 4.2. Safe operations can involve I/O, but they are implemented by Jovaku, with rigorous validation of arguments to minimize the potential for abuse.

Code Listing 4.2: Excerpt of the API for accessing cloud-side resources from a mobile function.

```
public interface IContext
{
    Task<List<string>> Get(string table, string key);

    Task<bool> Put(string table, string key, object value);

    Task<bool> Append(string table, string key, object value);

    Task<bool> Delete(string table, string key);

    Task<AmazonWebServiceResponse> Query(
        AmazonWebServiceRequest query);
}
```

To cover all intended use cases, we have tailored our `IContext` interface for interactions with generic key/value databases. Common operations such as *Get*, *Put*, *Append* and *Delete* are exposed through the interface, allowing interface developers to support different database systems through a common interface. Some cloud database systems offer a more elaborate interface to the database, with multiple keys, scan operations, or batch operations. Amazon DynamoDB is one of them, and to not reduce the functionality offered by the database, we extended the interface with the possibility to pass Amazon DynamoDB query objects directly. Similarly, when applying satellite execution with other cloud services, the `IContext` interface would be extended correspondingly, to expose the relevant functionality. Such functionality is not limited to low level database services, but could include composite services such as Facebook or Twitter as well.

The indirection created by the `IContext` interface also serves to separate application logic from the particulars of the cloud services that are accessed, and adds flexibility to deployments. For example, an application can be tested and run as a client-side process by providing a context object that binds to a local database.

4.2 Message Processor

From the relay-node's perspective, satellite execution entails processing a stream of incoming messages that contain serialized mobile functions. There must also be a protocol for transferring the .NET assemblies that contain the associated code. As detailed in Chapter 3, the relay-node is manifested as an instance in the EC2 computing cluster, where DynamoDB can be accessed with very low local network latency. The name server, which serves DNS traffic, is a BIND process with a custom extension implemented in C. Meanwhile, the message processor and execution environment must interface with .NET code. Therefore, these two layers in the architecture are implemented in C#, as a separate process that we refer to as the *execution server*.

We considered several approaches to implementing the message processor. One approach would be to use Windows Communication Foundation (WCF), a .NET framework for building service-oriented applications. Using WCF, developers can declare interfaces tagged with the `ServiceContract` attribute to expose them as a service to be accessed remotely. Specific methods to include are tagged with the `OperationContract` attribute, and WCF will wrap the implementations of these operations and expose them through a configured set of communication transports, such as HTTP or TCP. WCF hides the complexity of setting up listen sockets for the different communication

Code Listing 4.3: Interface for initializing the execution server with a specific underlying implementation.

```
public interface IExecutionServer
{
    bool StartListen(int port);

    bool StopListen();
}

public class ExecutionServerFactory
{
    public enum Implementation
    {
        WCF,
        APM,
        TAP
    }

    public static IExecutionServer Init(Implementation type) { ... }
}
```

transports, generating requests and replies, and managing concurrency. It also provides client libraries for accessing services asynchronously from other applications.

A more direct approach would be to manage sockets explicitly, and use a custom communication protocol to exchange messages over TCP. The main question in this approach is how to manage concurrency. The .NET framework provides several asynchronous programming patterns that make use of a managed thread pool, obviating the need to create separate blocking threads waiting for data on the client sockets. In earlier versions of .NET, the Event-Based Asynchronous Pattern (EAP) and Asynchronous Programming Model (APM) were the preferred ways to maximize concurrency and parallelism. EAP was introduced in .NET 2.0 and exposes events such as new connections, or data received on a socket. Handlers can be registered to process these events asynchronously from a thread pool. APM was introduced later, and centers around callback methods that are invoked when asynchronous operations complete. Again, a thread pool will drive the execution of callbacks. Version 4.0 of .NET included yet another innovation, with the Task Parallel Library (TPL). This introduces the Task Asynchronous Pattern (TAP), where asynchronous functions return awaitable tasks, obviating the need for event handlers and callback functions. TAP is currently the recommended approach to asynchronous programming in .NET.

Code Listing 4.4: The APM implementation of the execution server.

```
class APM_ExecutionServer : IExecutionServer
{
    public bool StartListen(int port)
    {
        tcpListener = new TcpListener(port);
        new Thread(Listen).Start();
        return true;
    }

    private void Listen()
    {
        tcpListener.Start();
        while (true)
        {
            tcpListener.BeginAcceptTcpClient(AcceptCallback, tcpListener);
            Wait();
        }
    }

    private void AcceptCallback(IAsyncResult asyncResult)
    {
        var newClient = tcpListener.EndAcceptTcpClient(asyncResult);
        Signal();
        HandleClient(newClient);
    }

    private void HandleClient(TcpClient client) { ... }
}
```

Our goal was to reduce latency as much as possible. Hence we were unsure which abstraction would provide us with the right opportunities to gain the insights we needed to reduce latency. To not miss any opportunities, we structured our implementation such that different communication mechanisms may be compared. Code Listing 4.3 shows the interface we defined for the execution server, and how the execution server is instantiated. One of several possible implementation types is passed to the `Init` method, in order to create an `IExecutionServer` instance. We developed concrete implementations of that interface using both the APM and TAP asynchronous patterns, along with a WCF implementation, so we could compare the new asynchronous pattern with the old one, and observe how asynchronous socket programming compares to WCF.

Both the APM and TAP implementations make use of the `TcpListener` class to create a listen socket for incoming connections, but how the socket is used

Code Listing 4.5: The TAP implementation of the execution server.

```
class TAP_ExecutionServer : IExecutionServer
{
    public bool StartListen(int port)
    {
        tcpListener = new TcpListener(port);
        new Thread(Listen).Start();
        return true;
    }

    private async void Listen()
    {
        tcpListener.Start();
        while (true)
        {
            var newClient = await tcpListener.AcceptTcpClientAsync();
            Task.Run(() => HandleClient(newClient));
        }
    }

    private async void HandleClient(TcpClient client) { ... }
}
```

differs quite significantly. Code Listing 4.4 outlines the message loop for the APM implementation. After the socket has been created and bound to the listen port, a thread is dedicated to accepting new connections. The accept thread enters a loop where it calls `BeginAcceptTcpClient` with a callback function that will be invoked from a separate thread (scheduled from a thread pool) as soon as there are new clients to accept. Only one accept operation can be in progress, so a condition variable is used to synchronize between the accept thread and the callback thread. In the callback function, `EndAcceptTcpClient` is first invoked on the listen socket, which will return a `TcpClient` object representing the newly accepted connection. The condition variable is then signalled, allowing the accept thread to initiate a new accept operation while the callback thread continues its execution in the `HandleClient` method. This method handles further communication over the client connection through multiple `BeginReceive` and `BeginSend` operations, along with their respective callback handlers.

The TAP implementation of the execution server, outlined in Code Listing 4.5, has a more intuitive and linear program structure. This is due to the `await` keyword, which allows asynchronous method invocations to be disguised as blocking operations. When `AcceptTcpClientAsync` is invoked, a task is returned that represents a commitment to produce a `TcpClient` object as soon

Code Listing 4.6: The WCF implementation of the execution server.

```
[ServiceContract]
interface IJovaku_WCF
{
    [OperationContract]
    byte[] HandleMobileFunction(byte[] mfData);

    [OperationContract]
    bool HandleAssembly(byte[] assemblyData);
}

class WCF_ExecutionServer : IExecutionServer
{
    public bool StartListen(int port)
    {
        var uri = string.Format("net.tcp://jovaku.com:{0}/", port);
        service = new ServiceHost(typeof(Jovaku_WCF), new Uri(uri));
        service.Open();
    }
}
```

as a new client has connected. The `await` keyword triggers a compile-time transformation of the method where it occurs, so that the result of the task can be awaited without blocking the calling thread. When a client connects, the task will complete and a thread from a thread pool will resume execution in the `Listen` method. Before initiating a new accept operation, the thread will schedule a new task that will invoke the `HandleClient` method, which will use the newly created `TcpClient` object for further communication. The implementation of `HandleClient` follows the same pattern to drive asynchronous I/O with what appears to be blocking operations.

The message loop in the WCF implementation is handled automatically by the WCF framework, so there is no explicit management of socket and connection state. The implementation, outlined in Code Listing 4.6, exposes a WCF service with two methods: `HandleMobileFunction` for processing individual mobile functions, and `HandleAssembly` for handling incoming assemblies. Clients send mobile functions by serializing them into a byte array that is passed to `HandleMobileFunction`. Similarly, they send assemblies by invoking `HandleAssembly` with the raw assembly data. The execution server hosts the WCF service by creating a `ServiceHost` object, which binds the WCF service to a URI that encodes the underlying transport protocol and port number.

The WCF client library provides synchronous and asynchronous method invo-

cation from the client, but there are no easy ways for the server to connect back to the client because WCF lacks support for hosting services on mobile devices. In contrast, the APM and TAP implementations manage sockets explicitly, and mobile devices can create listen sockets to receive connections from the server. This could be useful in scenarios where the connection is disrupted because of mobile network coverage issues, or if the client is requesting long running operations and does not want to wait for completion, but instead have the server connect back when the result is ready. A socket-based implementation also enables client-to-client communication, which could be relevant for future extensions.

Message Formats

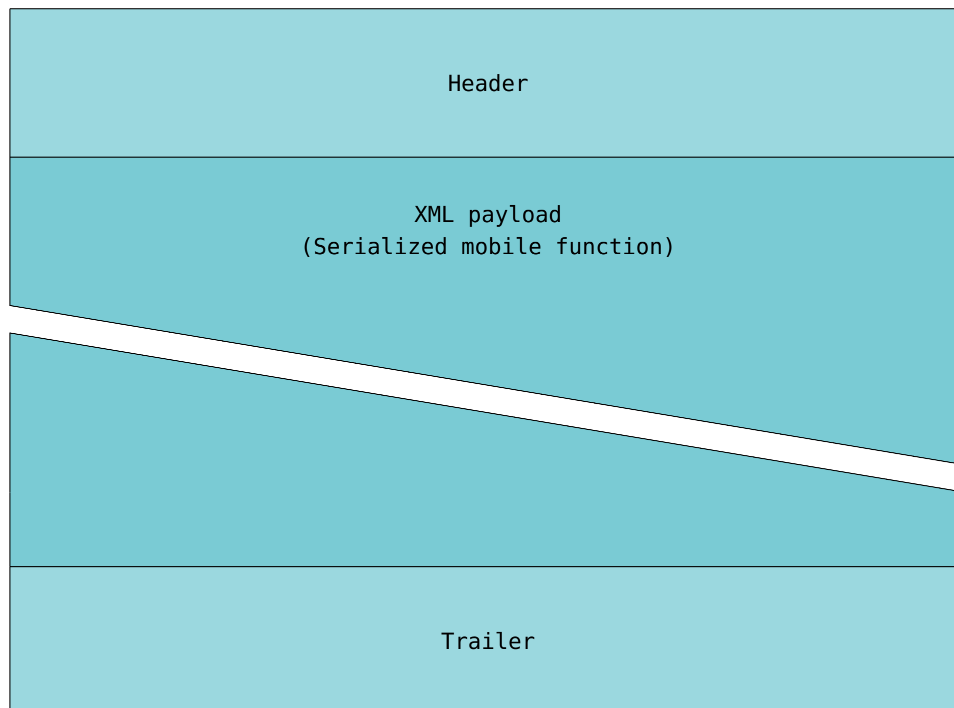


Figure 4.3: Layout of a WCF message, captured with WireShark when sending a mobile function containing four database operations. This message has a 68 bytes header, 818 bytes of XML payload, and a 79 bytes trailer, for a total of 965 bytes.

The choice of message processor implementation also affects the way messages are formatted. WCF handles communication and message formatting under the hood. To gain insight, we used Wireshark [112] to capture packets from the network for inspection while executing a sample application. Figure 4.3 shows an

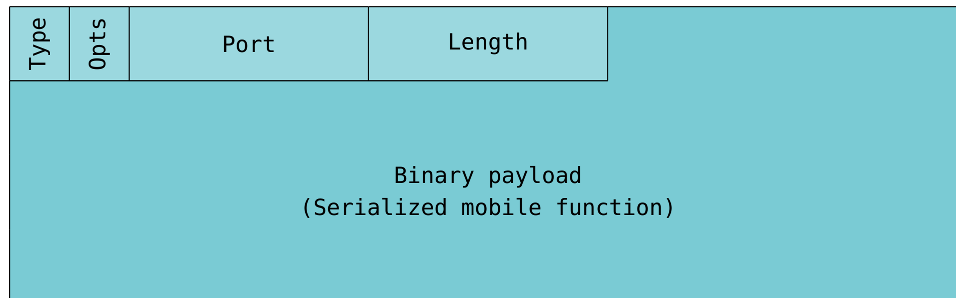


Figure 4.4: Layout of our own message format used in the APM and TAP implementations, representing the same mobile function as in Figure 4.3. This message has a 10 bytes header and 245 bytes of binary payload, for a total of 255 bytes.

example message, which represents one call to `HandleMobileFunction`.

WCF includes a serialization engine that is used for serializing parameters and return values into XML data. This XML data constitutes the main payload of the message. The standard serialization engine can be used instead with various formats; for example, the standard binary format is more compact and will reduce the overall message size.

In the socket-based APM and TAP implementations, we designed our own message format, shown in Figure 4.4. It consists of a header and a payload. The header has four fields: (1) the type of message, (2) a set of optional flags, (3) the client's listen port, and (4) the length of the payload. Depending on the message type, the payload is either a mobile function serialized in binary format, or raw data for an assembly.

For long-running mobile functions, the client can specify that the server should connect back after executing the mobile function by setting one of the optional flags. When this flag is set, the client can close the connection immediately after sending the request. The server will connect back to the client's listen port as soon as the mobile function completes, and send the reply over the new connection.

Although WCF comes with its own serialization engine, other engines can be used interchangeably in the different message loop implementations. The processing time required to produce serialized objects is generally negligible, regardless of the underlying engine. However, as message size can affect the network transfer time, smaller messages are preferred. Smaller messages can also reduce data transfer costs associated with a mobile plan.

Given that the TAP implementation has the most intuitive structure, allows us to host the message processor on mobile devices, and enables us to use a more compact message format, we decided to use this as the default implementation. All our experiments in Chapter 6 therefore use the TAP implementation. We have not observed any circumstances where the choice of message processor implementation has a major impact on performance.

4.3 Execution Environment

Incoming messages to Jovaku either contain serialized mobile functions that should be deserialized and executed, or .NET assemblies that contain the compiled code for mobile functions. Received assemblies are cached by Jovaku. Deserialization of a mobile function can fail if its assembly is missing. In that case, the client is asked to first send the missing assembly, before retrying. This will be a rare event in practical use, because mobile functions can be parameterized, reusing the same code across many instances, and because one assembly can contain the code for multiple mobile functions.

We sandbox the execution of mobile functions using .NET application domains [113], which provide an isolation boundary for security, reliability, and versioning, and for loading assemblies. Application domains are typically created by runtime hosts—which are responsible for bootstrapping the common language runtime before an application is run—but a process can create any number of application domains within the process to further separate and isolate execution of code. Jovaku creates a new application domain for each mobile function assembly.

Each of these application domains is configured with a minimal set of permissions that will ensure that execution of code cannot compromise or access code or data running in other domains. The minimal set will also ensure that the code received from clients cannot do potentially malicious operations like accessing and possibly deleting files from the file system, or participate in bot-nets that deplete network resources.

Code Listing 4.7 shows the code to instantiate new application domains. Aside from the minimal permission set, which only includes the most basic `Execution` ability, the code specifies a list of assemblies containing code that will be fully trusted by the sandbox. This includes Jovaku's own assemblies, and the official DynamoDB Software Development Kit (SDK) from Amazon.

When mobile functions execute, they can use the `IContext` interface to access cloud services. Jovaku implements this interface in the `SecureContext` class.

Code Listing 4.7: Creating a new application domain, with minimal permissions and set of trusted assemblies.

```
private Sandbox CreateSandbox(string name)
{
    var pSet = new PermissionSet(PermissionState.None);
    pSet.AddPermission(new SecurityPermission(Execution));

    var fullTrustAssemblies = new Assembly[]
    {
        typeof(Sandbox).Assembly,
        typeof(SecureContext).Assembly,
        typeof(Amazon.DynamoDBClient).Assembly,
    };

    var newAppDomain = AppDomain.CreateDomain(name, pSet,
        fullTrustAssemblies);

    var instance = Activator.CreateInstanceFrom(newAppDomain,
        typeof(Sandbox).Assembly.ManifestModule.FullyQualifiedName,
        typeof(Sandbox).FullName);

    return (Sandbox)instance.Unwrap();
}
```

Code Listing 4.8: Excerpt from the IContext implementation used in the isolated application domains.

```
class SecureContext : IContext
{
    private Amazon.DynamoDBClient _client;

    [SocketPermission(Assert, Unrestricted = true)]
    [ReflectionPermission(Assert, Unrestricted = true)]
    [WebPermission(Assert, Unrestricted = true)]
    public async Task<List<string>> Get(
        string table, string key) { ... }
}
```

The various database operations that can be performed are implemented using Amazon's SDK, which requires certain additional permissions to work correctly. Socket and web permissions are needed to create sockets and sending web requests, and the SDK uses reflection to access protected methods in the .NET library, to add custom headers to web requests. Since the assembly that implements `SecureContext` is fully trusted, these permissions can be elevated

Code Listing 4.9: The sandbox, isolating loading of untrusted assemblies, and execution of code.

```
class Sandbox : MarshalByRefObject
{
    private IContext Context;

    private Dictionary<string, Assembly> AssemblyCache;

    private Assembly AssemblyResolve(
        object sender, ResolveEventArgs args) { ... }

    public bool AddAssembly(byte[] rawBytes) { ... }

    public byte[] ExecuteFunction(byte[] obj) { ... }

    [SecurityPermission(Assert, Flags = SerializationFormatter)]
    private IMobileFunction DeserializeFunction(byte[] data) { ... }

    [SecurityPermission(Assert, Flags = SerializationFormatter)]
    private byte[] SerializeFunction(object graph) { ... }
}
```

selectively by marking the relevant methods with special security attributes, as illustrated in Code Listing 4.8. Therefore, the only way for a mobile function to access the network, for example, is through one of the methods of the `IContext` interface.

The `CreateSandbox` method returns a proxy object that can be used to communicate with the new application domain. Calls to the proxy object are implicitly converted into remote cross-domain calls. The `Sandbox` class in Code Listing 4.9 implements the internal execution environment of a sandbox, with methods to inject serialized assemblies and mobile functions into the sandbox. The sandbox will load the assemblies and put them into the `AssemblyCache` indexed on the full name of the assembly. The full name includes versioning information, so different versions of an assembly can be loaded at the same time.

Upon receiving serialized objects through the `ExecuteFunction` method, the sandbox will attempt to deserialize the byte array using the private method `DeserializeFunction`. This method is marked with a `SecurityPermission` attribute to allow deserialization of objects. We have restricted this permission to specific methods instead of allowing it for all client code, as the private data members of an object can potentially be retrieved by serializing it.

The sandbox also registers as a handler for the `AssemblyResolve` event, which is triggered whenever a new assembly must be resolved. Notably, this may happen during deserialization of mobile functions. The `ResolveEventArgs` will then contain the full name of the type that is being deserialized, and the sandbox can make lookups in the assembly cache to find the correct assembly. If the sandbox is unable to resolve the assembly required to deserialize the object, an exception will be thrown to the governing satellite execution environment, which in turn will inform the client of the missing assembly.

When an object has been deserialized successfully, the sandbox will typecast it to `IMobileFunction` and invoke its `Execute` method. When the `Execute` method completes, the mobile function is again serialized into a byte array, using `SerializeFunction`, and passed back to the client.

Mobile functions utilizing the `IContext` interface are similar to remote evaluation [114, 115] in many respects, but our implementation allows objects to be transmitted without accompanying code. By only transferring code when needed, we implement a form of lazy transfer that can realize further performance gains when code is rarely modified [116].

4.4 Client Library

To integrate satellite execution into Jovaku, we extended the main `IJovaku` interface as shown in Code Listing 4.10. From the client application's perspective, mobile functions are regular objects that may, upon request, be executed remotely. The `ExecuteAt` method in Code Listing 4.10 implements this abstraction by sending the object, in a serialized state, to a relay-node, where the object is deserialized and its `Execute` method is invoked. When the `Execute` method completes, the object is again serialized and moved back to the client. As such, mobile functions can simply store any relevant results of their cloud service interactions internally, and clients will be able to observe the corresponding state changes when `ExecuteAt` has completed.

The `location` argument to `ExecuteAt` specifies where to execute the mobile function. `GetExecutionEnvironments` allows clients to retrieve a list of Uniform Resource Identifiers (URIs) to different execution environments. The implementation will utilize the DNS protocol to locate connection information to the various execution environments instantiated in the zone that the client library is initialized with. The scheme of the URI will depend on the transport mechanism that the relay-node implements. A valid URI for an HTTP-based relay-node could be *http://relay.jovaku.com:8008/Jovaku*, while a socket-based relay-node could have the URI *net.tcp://relay.jovaku.com:8008/*.

Code Listing 4.10: Client side interface to utilize Satellite Execution.

```
public interface IJovaku
{
    // Methods from Code Listing 3.7 omitted

    List<Uri> GetExecutionEnvironments();

    Task<IMobileFunction> ExecuteAt(
        IMobileFunction function, Uri location = null);
}
```

Client applications implement mobile functions as classes implementing the `IMobileFunction` interface from Code Listing 4.1. Code Listing 4.11 shows an example mobile function that implements a simple *bag-of-queries*. Database queries may be be added to the bag by invoking `AddQuery` with the key that is to be queried for. Internally, the queries are collected in the `_queryList` field. `Execute` issues the queries to the database via the context object by iterating over the queries added by `AddQuery`. The results are stored in the `_responseList` field, which the client can retrieve by invoking `GetResponses`.

An object is moved for execution at a relay-node when the client application invokes the `ExecuteAt` method, specifying the object and the particular satellite execution environment. `ExecuteAt` transfers the object, in a serialized state, to the relay-node, where the object is deserialized and its `Execute` method is invoked inside a sandbox. After the `Execute` method completes, the object is moved back to the client, and the `ExecuteAt` method will return the object in its deserialized form.

A potential optimization for the bag-of-queries example would be to reset the list of queries to `null` once it is no longer needed. This would reduce the amount of serialized data to return from the relay-node to the client. In general, mobile functions are free to implement their own serialization mechanisms via the `ISerializable` interface, but they can always fall back to the default serialization protocol, for convenience.

By implementing a custom serialization mechanism, the transfer size can in some cases be reduced substantially. For example, the default serialization algorithm for the generic collection type `List` supports heterogeneous lists, where each item potentially has a different type. In many cases this is superfluous overhead, since the list is homogeneous and the item type is known in a priori. The custom serialization algorithm from Code Listing 4.12 is able to more than

Code Listing 4.11: Example implementation of a mobile function that provides a bag-of-queries abstraction.

```
[Serializable]
public class QueryBag : IMobileFunction
{
    private List<string> _responseList;
    private List<string> _queryList;

    public async Task Execute(IContext ctx)
    {
        foreach (var query in _queryList)
        {
            var queryResponse = await ctx.Get(query);
            if (_responseList == null)
                _responseList = new List<string>();

            _responseList.AddRange(queryResponse);
        }
    }

    public void AddQuery(string query)
    {
        if (_queryList == null)
            _queryList = new List<string>();

        _queryList.Add(query);
    }

    public List<string> GetResponses()
    {
        return _responseList;
    }
}
```

halve the resulting serialized byte array for a list of strings. As our experiments in Section 6.4 show, a similar optimization in a bag-of-queries with four queries reduced the resulting byte array from 700 bytes to 247 bytes.

Jovaku provides a location-independent programming abstraction, but preserves a monolithic application structure, which allows the application to be installed in its entirety on a single device through a regular distribution channel like an application store. Code is then transferred on demand from the device to the cloud, as objects move to the cloud to enjoy low-latency execution of database queries. The decision to visit the cloud or stay on the local device can be made dynamically, at runtime.

Code Listing 4.12: A custom serialization algorithm for a collection type containing strings. First the number of strings are stored, before the strings are added.

```
private void SerializeList(List<string> list, SerializationInfo info)
{
    var currentItem = 0;
    info.AddValue("Items", list.Count);
    foreach (var item in list)
    {
        info.AddValue(string.Format("Item {0}", currentItem++),
                      item);
    }
}
```

Traditionally, developers design and implement mobile applications in tandem with the corresponding cloud service. New use cases often lead to alterations or additions in the cloud service Application Programming Interface (API) to provide new functionality to the mobile application. To continue support for existing clients, APIs are often versioned to keep backwards compatibility. A study shows that developers can be reluctant to adopt new APIs [117], which leads to longer periods of time where a service provider needs to support old and deprecated API calls.

With satellite execution, new use cases are realized by introducing new mobile functions in the mobile application, instead of modifying the existing cloud service. This way, the mobile/cloud application evolves as a unit, rather than as two distinct entities. Updates to the functionality is distributed with the application, and many versions of the application can co-exist without explicit backwards compatibility in the cloud service.

Satellite execution has the potential to reduce latency for database operations, but will not provide additional guarantees—such as fault-tolerance—compared to executing the code directly on the device. It remains a responsibility of the developer to implement fault-tolerance in mobile functions if that is required.

4.5 Summary

In this chapter, we outlined an architecture with satellite execution designed to reduce completion-latency for a sequence of cloud database queries. Queries

are expressed as objects that interact programmatically with the database. Through satellite execution the objects are deployed in close proximity to the targeted cloud database. Figure 4.1 illustrates our approach, showing how multiple potentially expensive round-trips between a client and the cloud can be replaced with a single round-trip to the relay-node and multiple low-latency intracloud interactions with the database.

This approach preserves the advantages of a programmatic database interface; for example, objects can perform computations, transformations, cryptographic operations, and any other manipulations of arguments and intermediate results that may be required when performing a sequence of queries. However, we have not considered issues with fault tolerance beyond isolating execution of mobile functions in application domains. Connectivity issues with the mobile network, garbage collection of orphaned application domains, termination of endless mobile functions, and redundancy are important issues that need to be addressed before the system can be used in a production environment.

5 Applications

This dissertation focuses on deriving principles underlying the design of complex distributed software systems in order to improve their design and behaviour. Within such *systems research*, methods are experimental, emphasizing the construction, deployment and experimentation of actual software artifacts to substantiate conclusions. But these artifacts are not static once created. Rather, they are subject to a process of continuous refinement where experimental insights challenge assumptions and hypotheses, driving both incremental and radical changes to the design and implementation of the artifacts. The optimizations for mobile cloud interactions presented in this dissertation are the culmination of accumulated experiences whilst building and refining two proof-of-applicability prototypes: Picster and Dapper.

This chapter describes the design and implementation of Picster and Dapper. Both artifacts have undergone several iterations that have affected both their design and implementation, but the presentation focuses on how the artifacts, as they appear in their current and last version, have influenced and helped elicit the core contributions of this dissertation. Where appropriate and when serving to illustrate pivotal design choices, the design of early versions of the artifacts is highlighted.

5.1 Picster

Jovaku improves read latency from cloud databases by caching data at a Domain Name System (DNS) server in close proximity to the client. If the cached data is read by multiple clients, load to the cloud database is potentially further reduced, and sharing can help clients avoid initial cache miss overheads.

We envision many uses of Jovaku. One particularly good fit is applications that aim to be globally available, but whose primary function mostly involve read operations among users in close geographical proximity. Applications that use geographic services and capabilities to enable additional social dynamics are often referred to as geosocial networks. Examples of such networks are Yelp [118] and Around Me [119], which allow users to leave recommendations, experiences and reviews with the ultimate goal of helping other users make informed decisions about choosing, for instance, a restaurant or a hair stylist.

We developed Picster to explore this pattern of global availability and proximity-based client communication, so as to produce insights and input to the design of Jovaku. In particular, with Picster our goal was to drive and focus the aspects of Jovaku revolving around optimized cloud reads.

Picster is a collaborative image sharing, filtering, and ranking application for ad hoc geosocial networks. The application is centered around events where people come together and temporarily share common interests by virtue of their circumstances. For example, the users could be participants at a social gathering or spectators at a concert or a sports event. The application resembles iGroups [120], an ad hoc geosocial network from Apple that recognizes a need to optimize social communication when large groups of users gather in same spatial locality for a social event.

Traditional social networks use the cloud as a rendezvous point for locating other users, media sharing, and collaboration. In an ad hoc geosocial application, such as Picster or iGroups, rendezvous in the cloud would result in substantial network traffic from mobile devices to the cloud when videos or images are exchanged among spectators for viewing, ranking, and commenting. This is likely to cause unresponsive network connectivity [121].

With Picster, mobile users form an impromptu social network to exchange and rank pictures and comments. Through real-time collaborative filtering, the best pictures and most apt commentary is ranked and distributed among the users. While the social network can extend around the world, a majority of users are expected to be in geographical proximity to the event that sparked its formation. Beyond entertaining their participants, we imagine similar applications can be

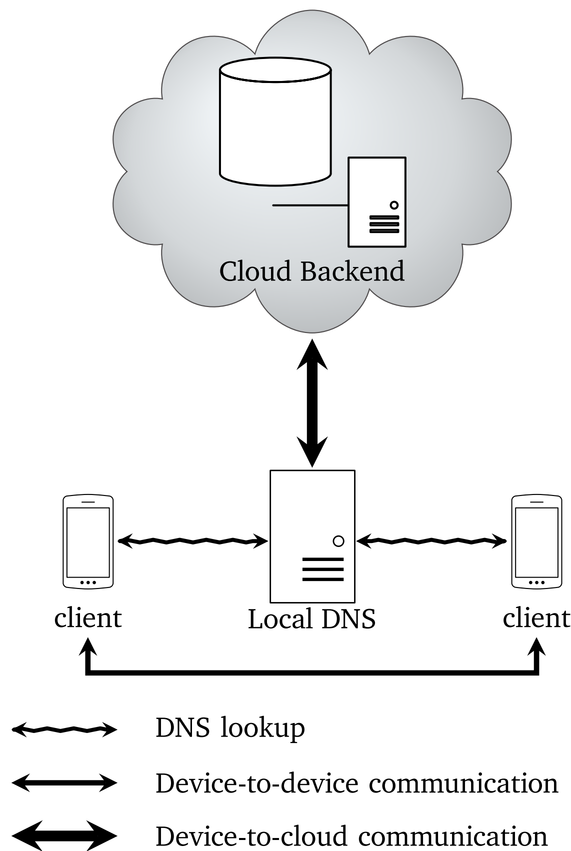


Figure 5.1: Picster social network architecture. Clients access the cloud backend service through the local DNS server. Image metadata and connection information to other clients is cached in the DNS server, enabling quick lookups for clients sharing the DNS server. Actual image data is exchanged directly between devices, reducing network load.

employed by broadcasters to enrich their products with high-quality content that is generated on the fly by the live audience.

Picster relies on a cloud database to store metadata such as comments, subscriptions, and membership information, and Jovaku is used as a caching layer to provide efficient and cheap access to the database for the common case where clients are in close proximity. An overview of the architecture can be seen in Figure 5.1. Picture data is exchanged directly between devices, while the metadata in the database is used as a simple and reliable way to locate and establish connections to peers. The existence of a shared, central database simplifies the architecture compared to alternatives like symmetric peer-to-peer architectures. In addition to reducing the load on the cloud database, Jovaku also mitigates the risk that bandwidth to the cloud becomes

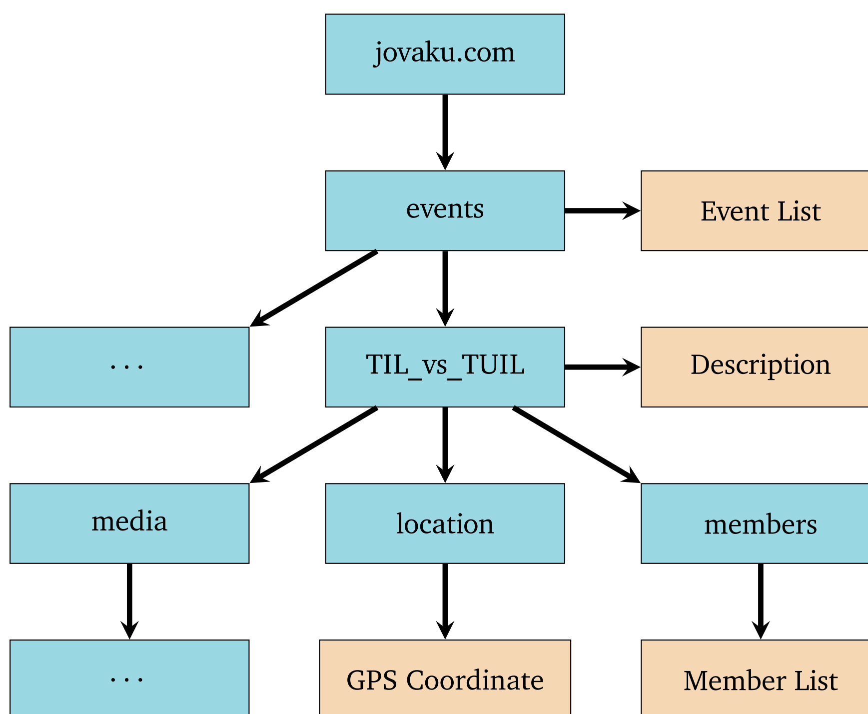
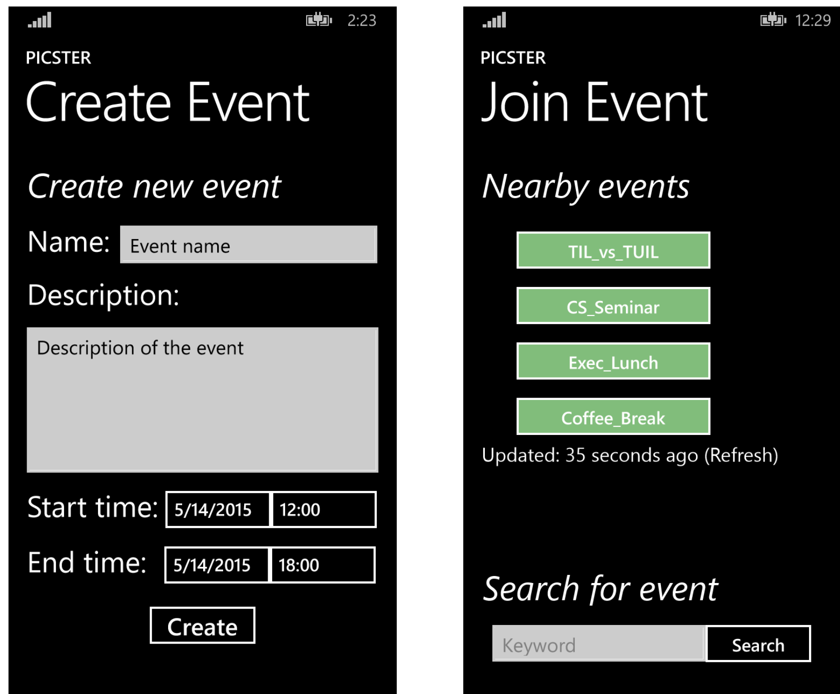


Figure 5.2: Domain name hierarchy illustrating an event "TIL_vs_TUIL", with its description, location and member list. Blue nodes illustrate labels, while yellow nodes describe the values associated with those labels.

a bottleneck [122].

Picster is centered around groups of people participating in the same event. Events can be searched by name or description, or by using GPS coordinates to discover nearby events. Creation of an event involves creating new labels in the domain name hierarchy to the domain hosting the events, in our case *jovaku.com*. Figure 5.2 illustrates an example DNS hierarchy of an event that has been created for the football match between TIL (Tromsø IL) and TUIL (Tromsdalen IL). The label *events.jovaku.com* can be queried for TXT Record Set (RS) to retrieve a list of available events, where *TIL_vs_TUIL* will be found.

The *TIL_vs_TUIL.events.jovaku.com* label contains a TXT record with description and meta information about the group. The *location* sub-label contains GPS coordinates for the event, which enable users to locate events they are close to, while the *members* label contains a RS with connection information for members of the event. A client joins an existing event by adding his connection information to the RS, and leaves by removing the information.



(a) Create a new event with description and a period when the event is occurring.

(b) Discover nearby events and search for others.

Figure 5.3: Creating and locating events in Picster.

Picster utilizes the Jovaku Software Development Kit (SDK) to perform database operations through DNS. These operations are exposed to the user through an intuitive user interface illustrated in Figure 5.3. As shown in Figure 5.3a, a user can create a new event by providing a name, description and a time span for event occurrence.

Figure 5.3b illustrates how the user can search and discover nearby events. By retrieving all events and their location sub-label, Picster can filter events that are in close proximity to the user. This list of events is cached by the local DNS server, and because standard DNS replies only contain the remaining time the values will be cached, the application has no way of determining how long the values have been cached already. Depending on the original Time To Live (TTL) value of the label, the user must wait until the cache expires before being able to acquire updated information.

Figure 5.3b illustrates how the user can search and discover nearby events. By retrieving all events and their location sub label, Picster can filter events that are in close proximity to the user. This list of events is cached by the local DNS

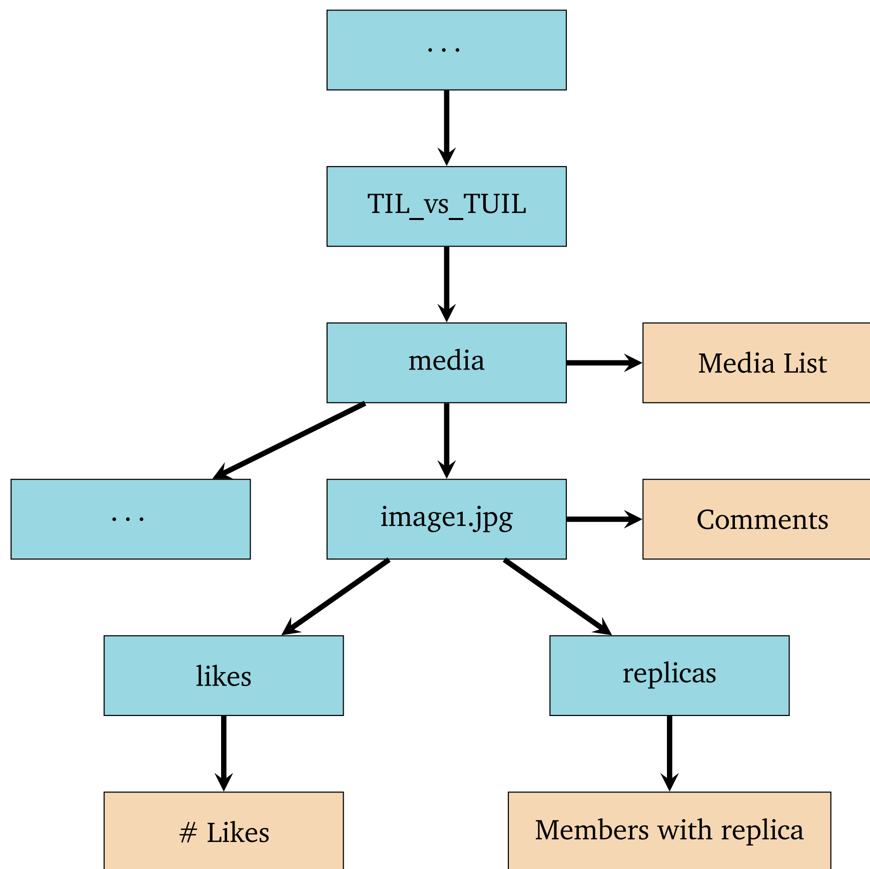
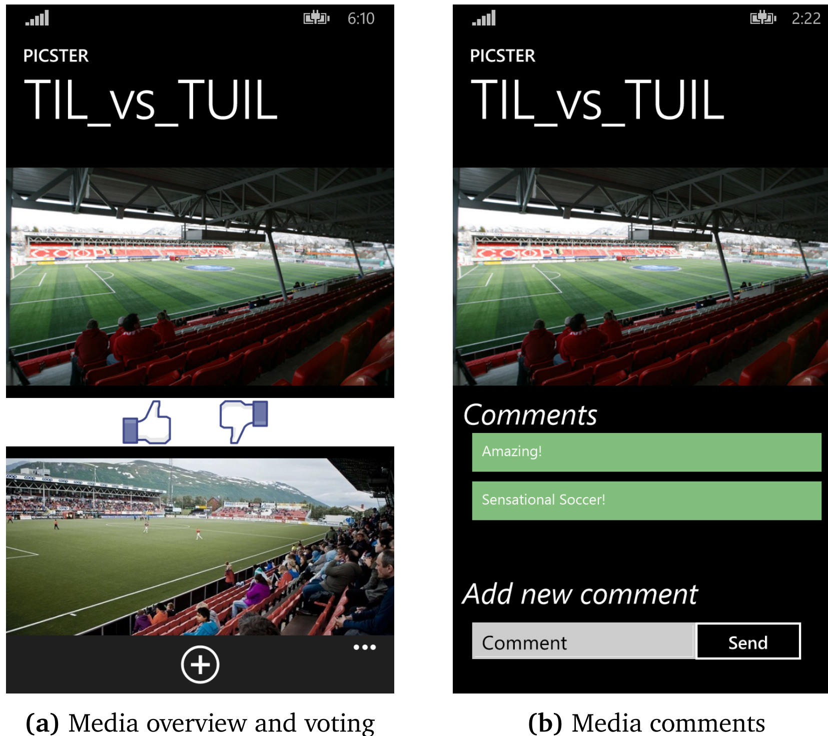


Figure 5.4: Domain name hierarchy illustrating the media tree under the "TIL_vs_TUIL" event. Blue nodes illustrate labels, while yellow nodes describe the values associated with those labels.

server. An important design input to Jovaku was derived from this scenario. Consider that standard DNS replies only describe when a value expires and *not* the time at which the value was inserted into the cache. Thus, a returned value might not be temporally valid, failing to convey e.g. the cancellation of an event. The problem is addressed by Jovaku through prefixing all values with the original TTL, enabling an application to determine cache durations and possibly request refreshes by forcing the local DNS server to omit the cache and retrieve updated values from the authoritative DNS server if demanded by the scenario.

After joining an event, members can add new photos to the event, either by capturing a new photo within Picster, or by browsing to an existing image on the device. Adding new photos to an event will update the DNS hierarchy similar to adding an event. Figure 5.4 shows the resulting *media* sub-tree under the TIL_vs_TUIL event after adding an image. The *media* label contains TXT



(a) Media overview and voting

(b) Media comments

Figure 5.5: The Picster application, which stores image metadata in a cloud database, using Jovaku for caching.

RSs with a list of available images for the event, where *image1.jpg* will be found. The TXT record for this label contains comments for the image. Below all images we find two labels, *likes* and *replicas*. The label *likes* contains the number of likes images have received, and the label *replicas* contains connection information to members that host actual image data.

To speed up the dissemination of new images, a rumor-mongering gossip protocol [123] is used to advertise new content to other members of the event. The protocol is based on user evaluation of the content. If media content receives a *like* from a member, the application will spread information to a random selection of other members. Thus, liked content will with high probability continue to spread to all members, and content that is not liked will eventually disappear.

Figure 5.5a shows how Picster presents new images to users, enabling collaborative filtering by allowing users to like or dislike images. Liking an image will store it on the device, allowing retrieval from other members. The *replicas* label will be updated, and a gossip message will issued to help dissemination. Clicking an image allows users to see comments from other members, and add new

Figure 5.6: The Picster web application displaying an image from the TIL_vs_TUIL event, with the number of likes and a list of comments.

comments. Figure 5.5b shows an image and its associated comments.

Picster was developed as part of a broader project that investigates decentralized deployment of cloud services [45, 46] in the context of sports analytics. As part of this research, the stadium and players of a premier soccer club have been equipped with various sensor and video processing technologies. During matches, coaches interface with the system in real time through mobile

devices [51, 53] to capture important in-game events.

To involve spectators, we apply Picster to identify events with high entertainment value. To present high quality content at an event, we developed an ASP.NET application called *PicsterWeb*, shown in Figure 5.6. PicsterWeb was designed to run on large presentation screens at events, but can also be accessed as a normal web page. PicsterWeb can join events as a spectator and keep track of images that have received likes above a given threshold. The resulting images are retrieved and stored locally, and can be viewed in a web browser or presented on a large screen at the event.

5.2 Dapper

Our goal with the Jovaku middle tier is to provide functionality to reduce latency when a mobile/cloud application communicates with the cloud. The offered functionality ranges from caching of cloud data in close proximity to a client, as described in Chapter 3, to moving code from the client to the cloud through satellite execution, as described in Chapter 4.

Whether the features of Jovaku are straightforward to exploit for mobile/cloud applications is a question that, following a systems research method, should be answered through construction of artifacts. The Picster artifact substantiates that ad hoc geosocial applications can derive benefits from utilizing the Jovaku caching functionality. We developed the Dapper artifact to investigate and evaluate Jovaku's satellite execution concept.

Dapper is a social network application that implements the traditional cloud backend service as mobile functions in the application. With Dapper, users can create a profile and post status updates to a feed. Users can also search other users of the social network and send a request to add them to their friend list. The functionality of Dapper closely resembles that of the core functionality of Facebook, albeit in a simpler form. Dapper seeks to validate whether satellite execution can be used to construct mobile/cloud applications.

Implementing code that interacts with the cloud database as mobile functions allows Dapper to take advantage of satellite execution in Jovaku. By implementing a local context object, latency saving can be measured for each of the mobile functions implemented in Dapper.

Dapper uses three database tables: Profiles, Friends and Feed. The Profiles table contains all user profiles, allowing users to maintain a profile and enable searches for friends. The Friends table contains a list of all friends and pending

friend requests. The Feed table contains social updates pertaining to a user. The layout of the database tables can be seen in Tables 5.1, 5.2 and 5.3, and will be detailed shortly.

Table 5.1: The *Profiles* table contains profiles for users of Dapper, allowing others to locate friends by searching for name or description.

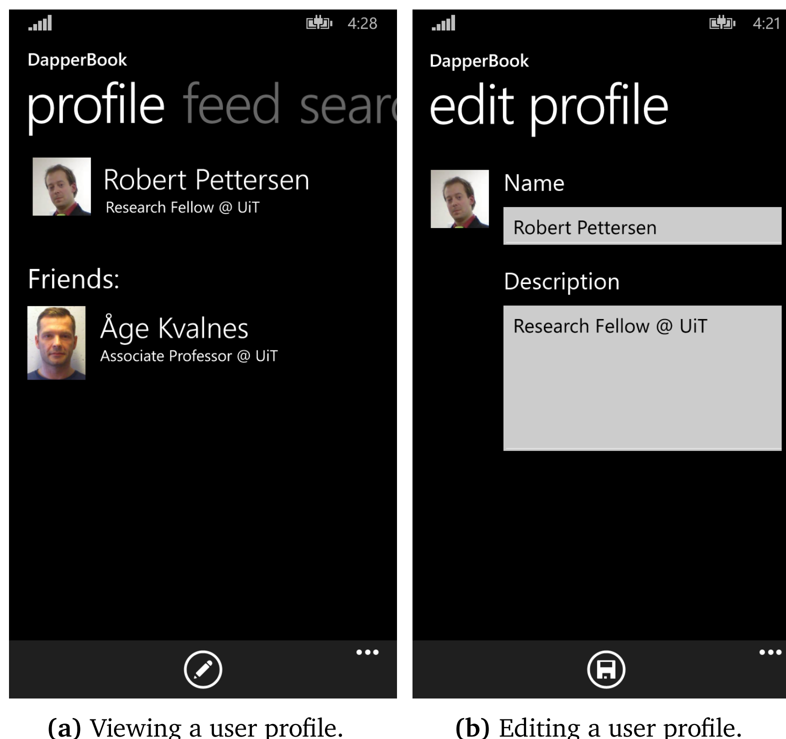
UserId	Timestamp	Name	Description
USER1...	1446905829	Robert Pettersen	Research Fellow @ UiT
USER2...	1446903159	Åge Kvalnes	Associate Professor @ UiT
USER3...	1446901279	Dag Johansen	Professor @ UiT

Table 5.2: The *Friends* table contains friend relationships and pending friend requests. The table contains one friend relationship and a pending friend request.

UserId	Timestamp	FriendId	Relationship
USER1...	1446905829	USER2...	FRIEND
USER2...	1446905829	USER1...	FRIEND
USER3...	1446907537	USER1...	REQUESTED
USER1...	1446907537	USER3...	PENDING

Table 5.3: The *Feed* table contains all social updates pertaining to a user, which includes updates from close friends and changes to the friend list in addition to updates posted by the user itself.

UserId	Timestamp	Message	Origin
USER1...	1446905829	Found a bug!	USER2...
USER1...	1446904735	Became friends with Åge	FRIEND
USER1...	1446903159	Brr freezing outside	USER1...
USER1...	1446901279	Winter is coming	USER1...



(a) Viewing a user profile. **(b)** Editing a user profile.

Figure 5.7: Handling user profiles in Dapper.

5.2.1 User Profile

To uniquely identify users of Dapper, the anonymous identifier obtained from the `UserExtendedProperties` class in the Windows Phone SDK is used. This reduces complexity of the application by removing the need to register new accounts and authenticating existing users, allowing us to focus on functional aspects.

The layout of the profile table can be seen in Table 5.1, and contains four fields: UserId, Timestamp, Name and Description. The UserID field is populated from the anonymous identifier, but is displayed here in a human readable format for illustrative purposes. The Name and Description fields are maintained by the user and will be used to locate other users of the application.

Users view their profile under the profile tab in the application, illustrated in Figure 5.7a. A list of friends will also be displayed, populated as social connections are made. Changes to the profile can be made by clicking the edit button on the bottom of the page, and making changes as necessary. The edit profile page is shown in Figure 5.7b and is stored by clicking the save button.

Code Listing 5.1: The `ProfileUpdate` mobile function will update a user profile, or create a new one if the profile does not exist.

```
[Serializable]
public class Profile
{
    public string UserId { get; protected set; }
    public DateTime Timestamp { get; protected set; }
    public string Name { get; set; }
    public string Description { get; set; }
}

[Serializable]
public class ProfileUpdate : Profile, IMobileFunction
{
    public async Task Execute(IContext ctx)
    {
        Timestamp = DateTime.Now;
        var rawProfile = await ctx.Get("Profiles", UserId);
        if (rawProfile == null)
        {
            await ctx.Put("Profiles", UserId, this);
        }
        else
        {
            var profile = ParseProfile(rawProfile);

            profile.Timestamp = Timestamp;
            if (Description != null)
                profile.Description = Description;
            if (Name != null)
                profile.Name = Name;

            await ctx.Put("Profiles", UserId, profile);
        }
    }

    private Profile ParseProfile(string data) { ... }
}
```

The `Profile` class is used to represent a user profile, and mobile functions that operate on user profiles inherit this base class to get the required properties. The `ProfileUpdate` mobile function seen in Code Listing 5.1, is used both to create new profiles and update existing ones. If the user does not have a profile, a new one will be created, and if there is an existing profile, altered fields will be updated with new values.

5.2.2 Connecting with Friends

Table 5.2 shows the layout of the Friends table, and contains four fields: UserId, Timestamp, FriendId and Relationship. The Relationship field describes the relationship between UserId and FriendId. From the table we observe a friendship between *USER1* and *USER2* denoted by the *FRIEND* relationship, and a friend request from *USER3* to *USER1* denoted by *REQUESTED* and *PENDING* relationships.

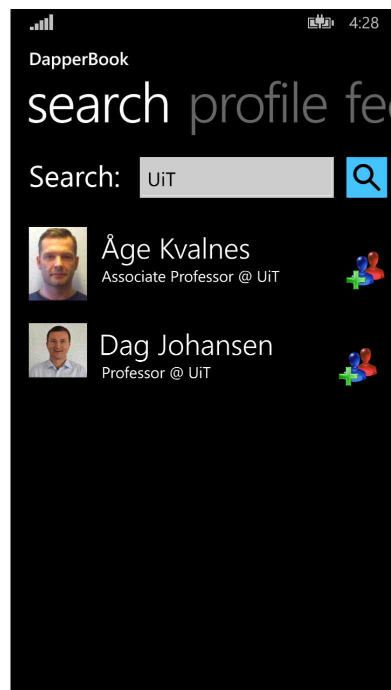


Figure 5.8: Searching for friends in Dapper.

Searching for friends in Dapper is performed under the search tab, illustrated in Figure 5.8. The search is performed by a keyword search spanning the Name and Description fields in the Profiles table, allowing searches both for name and description. The list of matches will be presented to the user, which in turn can choose who to send friend requests to.

Sending a new friend will invoke the `FriendRequest` mobile function, shown in Code Listing 5.2. `FriendRequest` inherits from the `Friend` class that represents a social connection, providing the required properties. The mobile function will create new entries in the *Friends* table, declaring the request. First a new entry for the user sending the request will be added, with the relationship set to *REQUESTED*. Then an entry for the receiving user is added with the relationship set to *PENDING*.

Code Listing 5.2: The `FriendRequest` mobile function will insert a new pending friend request into the *Friends* table.

```
[Serializable]
public class Friend
{
    public string UserId { get; protected set; }
    public DateTime Timestamp { get; protected set; }
    public string FriendId { get; set; }
    public string Relationship { get; set; }
}

[Serializable]
public class FriendRequest : Friend, IMobileFunction
{
    public async Task Execute(IContext ctx)
    {
        Timestamp = DateTime.Now;

        Relationship = "REQUESTED";
        await ctx.Put("Friends", UserId, this);

        Swap(UserId, FriendId);
        Relationship = "PENDING";
        await ctx.Put("Friends", UserId, this);
    }
}
```

5.2.3 Status Updates

The layout of the *Feed* table can be seen in Table 5.3, and contains four fields: UserId, Timestamp, Message and Origin. The UserId field points to the owning user, Message contains the content and Origin specifies where the message originated from. The Origin field can either be FRIEND to describe changes to the friend list, or a user id pointing to the creator of the status message.

The social feed can be found under the feed tab in Dapper, shown in Figure 5.9. The different types of messages are differentiated with icons, where blue speech bubbles are the users own updates, green are updates from friends and the people icon indicates changes to the friend list.

Posting a new status message executes the `StatusUpdate` mobile function, shown in Code Listing 5.3. The mobile function starts by posting the status to the poster's own feed. Then the friends of the poster is retrieved, and each friend is subjected to an algorithm to determine whether the status should

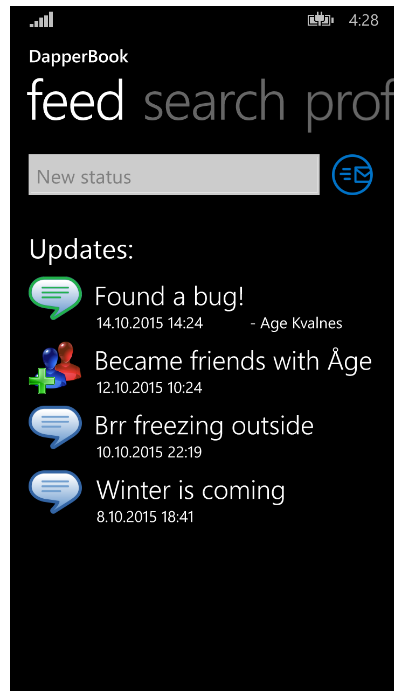


Figure 5.9: The feed shows social updates from the user and its closest friends, and changes to the friend list.

be posted to their feed as well, by invoking `IsRelevantFor`. Possible elements that factor into the decision could be number of likes, keywords in the status message or relationship status. For the moment the algorithm only considers relationship status.

5.2.4 Making Progress

So far Dapper can create and make changes to a user profile, post status updates and send friend requests. To retrieve updates and friend requests, Dapper has a periodic background task that executes the `RetrieveUpdates` mobile function, shown in Code Listing 5.4. The mobile function will start by retrieving the feed for the user, and filter out status messages that are older than the last run. The remaining status messages are added to a list, and will be merged with the existing status message list when the mobile function returns.

After new status updates have been retrieved from the database, the friend table is checked for pending friend requests. If there are any pending friend requests, the profile of the requesting user is retrieved to give enough information to aid in deciding whether or not this is a friend. The profile is wrapped in

Code Listing 5.3: The `StatusUpdate` mobile function will update a user's status by posting the new status to relevant feeds.

```
[Serializable]
public class StatusUpdate : IMobileFunction
{
    public string UserId { get; private set; }
    public DateTime Timestamp { get; private set; }
    public string Message { get; set; }
    public string Origin { get; set; }

    public async Task Execute(IContext ctx)
    {
        Timestamp = DateTime.Now;
        Origin = UserId;
        await ctx.Append("Feed", UserId, this);

        var friends = ParseFriends(await ctx.Get("Friends", UserId));
        foreach (var friend in friends.Values)
        {
            if (IsRelevantFor(friend))
                await ctx.Append("Feed", friend.FriendId, this);
        }
    }

    Dictionary<string, Friend> ParseFriends(string data) { ... }
    bool IsRelevantFor(Friend friend) { ... }
}
```

a new `PendingFriend` object, before adding it to the list of pending friend requests.

The retrieved profile is cast to the `FriendRequest` mobile function, seen in Code Listing 5.5. `FriendRequest` inherits from the `ProfileUpdate` class giving it all the necessary properties to contain the profile. In addition to the profile fields, the `FriendRequest` mobile function has a property indicating whether or not the user has accepted the request.

The `PendingFriend` object is a mobile function itself, shown in Code Listing 5.5. If the request is accepted, the mobile function will be executed. The function will start by retrieving both requests from the Friends table. The Relationship is changed to *FRIEND* to indicate a normal friendship, and stored in the Friends table. The function will then advertise the new friendship to both users' feed.

Code Listing 5.4: The `RetrieveUpdates` mobile function will retrieve social updates and pending friend requests.

```
[Serializable]
public class RetrieveUpdates : IMobileFunction
{
    public string UserId { get; private set; }
    public List<StatusUpdate> Feed { get; private set; }
    public List<PendingFriend> PendingFriends { get; private set; }
    public DateTime LastUpdated { get; set; }

    public async Task Execute(IContext ctx)
    {
        // Retrieve previously unseen social updates
        var statusUpdates = await ctx.Get("Feed", UserId);
        foreach (var update in statusUpdates.
            Where(s => s.Timestamp > LastUpdated))
        {
            Feed.Add(update);
        }

        // Retrieve pending friend requests
        var friendList = await ctx.Get("Friends", UserId);
        foreach (var friendRequest in friendList.
            Where(f => f.Relationship == "PENDING"))
        {
            var friendProfile = await ctx.Get("Profiles",
                friendRequest.FriendId);

            PendingFriends.Add(new PendingFriend
            {
                UserId = UserId,
                fProfile = friendProfile,
            });
        }
    }
}
```

5.2.5 Experiences and lessons learned

Dapper uses the satellite execution capability in Jovaku to implement a mobile social networking application. Mobile functions are used to update user profiles, search for and add friends, and post status updates by moving code from the application to the cloud. Developing mobile functions for satellite execution can be compared to developing a cloud side Application Programming Interface (API) that users of the service can utilize in applications.

Code Listing 5.5: The `FriendRequest` mobile function will accept a friend.

```
[Serializable]
public class PendingFriend : IMobileFunction
{
    public string UserId { get; set; }
    public string Name { get; set; }
    public Profile fProfile { get; set; }

    public async Task Execute(IContext ctx)
    {
        // Get user's friends
        var uFriends = ParseFriends(await ctx.Get("Friends",
            UserId));

        // Get friend's friends
        var fFriends = ParseFriends(await ctx.Get("Friends",
            fProfile.UserId));

        // Set the relationships to FRIEND
        fFriends[UserId].Relationship = "FRIEND";
        uFriends[fProfile.UserId].Relationship = "FRIEND";

        // Update friends table
        await ctx.Put("Friends", fProfile.UserId, fFriends);
        await ctx.Put("Friends", UserId, uFriends);

        // Update feeds to advertise the relationship
        var statusUpdate = new StatusUpdate { Origin = "FRIEND" };

        statusUpdate.Message = string.Format(
            "Became friends with {0}", fProfile.Name);
        await ctx.Append("Feed", UserId, statusUpdate);

        statusUpdate.Message = string.Format(
            "Became friends with {0}", Name);
        await ctx.Append("Feed", fProfile.UserId, statusUpdate);
    }

    Dictionary<string, Friend> ParseFriends(string data) { ... }
}
```

Using mobile functions in Dapper is similar to how the Facebook API is used in Android. Code Listing 5.6 compares how mobile functions are used to post status updates in Dapper, with how the official Android Facebook API is used to send status updates to Facebook. The main difference lies in where the logic that handles the request is located. With Jovaku the logic is contained in mobile functions, while Facebook has the logic in their datacenters.

Code Listing 5.6: Comparison of posting status updates to Dapper using Jovaku and using the Facbook API to post updates to Facebook.

```
// Using satellite execution to post status updates to Dapper
var update = new StatusUpdate();
update.Message = "Hello World!"
await JovakuClient.ExecuteAt(update, RelayNode);
/* handle the result */


// Using Facebook API to post status updates to Facebook
Bundle param = new Bundle();
param.putString("message", "Hello World!");
/* make the API call */
new GraphRequest(
    AccessToken.getCurrentAccessToken(),
    "/me/feed",
    param,
    HttpMethod.POST,
    new GraphRequest.Callback() {
        public void onCompleted(GraphResponse response) {
            /* handle the result */
        }
    }
).executeAsync();
```

When the mobile functions in Dapper evolved during development, we met one important challenge. When multiple versions of the same code was loaded in the relay-node at the same time, conflicts occurred when trying to determine the correct data type. Supporting multiple versions of the same mobile functions is analogous to API evolution in traditional cloud services. Changing an API that clients and applications use must be done with care, as developers are often slow to adopt API changes [124].

There are three overall models for API evolution [125]: The Knot model, Point-to-point model and Compatible versioning. The Knot is the simplest form of evolution, where all API consumers are tied to a single version of the API. Changes in this model creates a massive ripple effect across consumers. In Point-to-point evolution, every API version is left running in production and consumers can migrate on their own, when they need to. Compatible versioning means that all changes to an API must be compatible with the previous version.

Point-to-point and Compatible versioning both lead to higher maintenance cost for the API provider, since multiple versions of the API need to be maintained.

By using separate application domains for different versions of an assembly, the relay-node is able to host multiple versions at the same time. Because code comes from mobile applications, API evolution and maintenance is minimized.

Applications running on mobile devices have to explicitly request access to resources such as networking, contact lists, and file access. The ability to restrict access to resources similarly when executing code in the cloud is therefore desirable. Application domains have the ability to restrict access to resources for code running inside the isolated container. Another side effect of introducing application domains is increased fault tolerance for the relay-node, since faults pertaining to one assembly is isolated in the application domain and will not affect the rest of the relay-node, as described in Section 4.3.

5.3 Summary

In this chapter we present two mobile social network applications, Picster and Dapper, that both leverage Jovaku to reduce communication latency. Picster exploits the close proximity to DNS servers to reduce read latency to data that multiple users in close vicinity access. Dapper reduces write latency by serializing code and objects for remote execution in the cloud, obviating the need to do multiple round-trips to complete a task. In the next chapter, the effectiveness of Jovaku will be evaluated through extensive experimentation.

6

Experimental Evaluation

To investigate our thesis that a generic middle tier can leverage existing infrastructure to reduce latency for mobile/cloud applications, we first designed and implemented Jovaku, as described in Chapters 3 and 4. This chapter experimentally evaluates the efficacy of Jovaku, measuring its performance and usefulness for mobile/cloud applications.

Jovaku can cache cloud database values close to the clients by exploiting the proximity of Domain Name System (DNS) servers to mobile clients, potentially reducing latency for lookups in mobile/cloud applications. Jovaku can also serialize code and data, and request remote execution to reduce latency for tasks that employ sequences of dependent queries.

To evaluate whether Jovaku can be used to reduce communication latency in mobile/cloud applications, we seek to answer the following questions:

> **Question 1.** Is DNS a viable placement for a caching middle tier?
>
> **Question 2.** Can modern mobile/cloud applications benefit from satellite execution in their operation?
>
> **Question 3.** Is satellite execution effective for reducing the latency of dependent queries?

Affirmative answers to the above questions would experimentally corroborate the efficacy of Jovaku to reduce communication latency in mobile/cloud applications. Assuming affirmative answers, an interesting question is then how well the relay-node scales as the number of clients grows. A fourth question that we aim to answer in our evaluation is therefore:

> **Question 4.** How does the relay-node scale as the workload increases?

To answer Question 1, we focus in particular on two key performance metrics. First, we seek to identify the worst case overhead of accessing DynamoDB by going through Jovaku. Second, we seek to identify the theoretical best case performance when all requests hit the Jovaku cache. In combination, the worst case and the best case performance measurements allow us to extrapolate the cache hit rate required to break even in terms of average request latency from a given location.

For Question 2, we need to investigate whether existing mobile/cloud applications make use of dependent queries. Since source code is unavailable and communication is encrypted, we conduct a black-box examination of communication patterns and seek to reveal patterns consistent with sequences of dependent requests.

To answer Question 3, we approximate the savings that could be experienced in a deployed application by comparing the execution of mobile functions with a varying number of dependent queries, with and without satellite execution.

Finally, to answer Question 4 we set up an experiment where an increasing number of mobile clients attempt to stress the relay-node as much as possible, by issuing mobile functions in a closed loop. Repeating the experiment with different resource allocations for the relay-node could reveal scaling properties of the relay-node.

6.1 Experimental Setup

Jovaku runs on a variety of Microsoft Windows platforms, including Windows Phone, Windows Store, and the traditional Windows desktop. We use four different platforms during the experiments: (1) a phone with 2 GB memory and a quad-core QualComm Snapdragon 800 2.2 GHz CPU, (2) a desktop machine with 64 GB memory and a quad-core Intel Xeon E5-1620 3.7 GHz CPU,

Table 6.1: Machine types used throughout the experimental evaluation, along with labels used to reference them.

Label	CPU type	CPU	Cores	Memory
Mobile	Snapdragon 800	2.2 GHz	4	2 GB
Desktop	Intel Xeon E5-1620	3.7 GHz	4	64 GB
EC2 t1.micro	64 bit vCPU	1.85 GHz	1	613 MB
EC2 t2.medium	64 bit vCPU	2.5 GHz	2	4 GB

(3) an Amazon EC2 t1.micro instance equipped with 613 MB memory and a single-core 1.85 GHz 64 bit vCPU, and (4) an Amazon EC2 t2.medium instance equipped with 4 GB memory and a dual-core 2.5 GHz 64 bit vCPU.

The phone runs Windows Phone 8.1 and communicates over 4G, whereas the desktop machine runs Windows 10 and is connected to a 100 Mbit/s LAN. Both EC2 instances are running Microsoft Windows Server 2012 R2. The phone and desktop are located in Tromsø, while the EC2 instances are instantiated in various Amazon availability zones, detailed further in each experiment. An overview of the machine types involved in the experimental evaluation can be seen in Table 6.1, along with labels that we will use to reference them henceforth.

6.2 DNS caching

The effectiveness of caching depends on the ratio of cache hits, which again depends on the access pattern of the application. We envision diverse applications for Jovaku, that may exhibit many different access patterns. Performance also depends on how an application is deployed geographically, since this affects the latency to access both DynamoDB and DNS. Therefore, we use synthetic workloads in our experiments, and run experiments from multiple locations, so we could predict performance for a range of cache hit rates and sites across the world.

We use Amazon's availability zone in Ireland to host both the relay-node and the DynamoDB service. Table 6.2 shows the machines used in these experiments, along with the labels we will use to reference them henceforth. For each location, the fourth column lists typical ping latency (the average of 1000

Table 6.2: Machines involved in evaluating the effectiveness of DNS caching, along with the latency and hop count to the Amazon DynamoDB located in Ireland.

Label	Location	Type	Ping Latency	# Hops
Norway	Tromsø	Desktop	64 ms	15
Norway-3G	Tromsø	Desktop	116 ms	13
Norway-4G	Tromsø	Desktop	120 ms	18
US West	California	EC2 t1.micro	155 ms	17
Asia	Singapore	EC2 t1.micro	339 ms	20
Australia	Sydney	EC2 t1.micro	365 ms	12
Relay-node	Ireland	EC2 t1.micro	4 ms	3
Relay-node#2	Ireland	EC2 t2.medium	4 ms	3

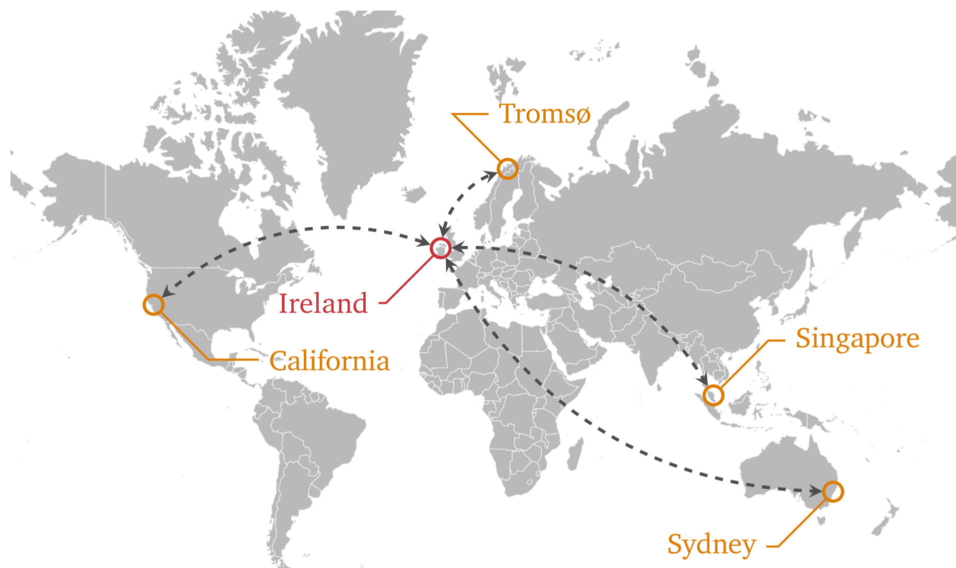


Figure 6.1: Placement of nodes on world map. Client locations chosen to exhibit some variations in routing distance and latency to the DynamoDB service and relay-node located in Ireland.

pings) and the fifth column lists the number of routing hops to DynamoDB according to *traceroute*. We use mostly EC2 instances from Amazon's various availability zones, complemented with a desktop machine in Norway. To access mobile networks, we use a USB dongle to access the 3G network operated by Netcom, and Wi-Fi tethering to access the 4G network operated by Telenor. Figure 6.1 shows the various geographical locations of the machines, chosen to exhibit some variation in routing distance to the DynamoDB service and relay-node in Ireland.

We focus in particular on two key performance metrics. First, we seek to identify the worst case overhead of accessing DynamoDB by going through Jovaku. To this end, we measure the baseline performance of accessing DynamoDB directly, compared to the performance of using Jovaku with a workload that never hits the cache. In this worst case scenario, each lookup is a cache miss as depicted in Figure 3.1b. The lookup must first consult the local DNS server, before it is forwarded to the authoritative DNS server, which is our relay-node in the cloud, where it is translated into a database lookup using the DynamoDB Application Programming Interface (API). We provoke this worst case behavior by setting the Time To Live (TTL) of the database values to 0. This disallows caching by local DNS servers, forcing every request to be forwarded to the authoritative source.

Second, we seek to identify the theoretical best case performance when all requests hit the Jovaku cache. In this case, every lookup is for a key that has previously been cached and resides in the local DNS server. Requests only incur the latency of a local DNS lookup and never go all the way to the cloud. We induce this behavior by setting the TTL of our database values to 24 hours. We then prime the local DNS cache by making several lookups for all values. This is required since we use Virtual Machines (VMs) in various EC2 clusters as clients, and these VMs are configured to multiplex DNS requests over a large set of local DNS servers. A warm up period is thus required to populate all the DNS caches. In deployments where clients are actual mobile devices around the world, rather than VMs in a data center, these kinds of DNS setups would be rare to encounter.

In combination, the best and worst case performance measurements allow us to extrapolate the exact cache hit rate required to break even in terms of average request latency from a given location. Applications whose hit rates are expected to be above the break-even threshold will thus benefit from using Jovaku as a caching layer. In addition, every single cache hit also reduces the load on the cloud database service, and saves money when services are billed per request, as is the case with DynamoDB. In other words, applications that are less sensitive to latency may benefit from Jovaku even for cache hit rates below the break-even threshold.

6.2.1 Baseline Performance

Our first set of experiments establishes the baseline performance of accessing DynamoDB in Ireland. For each client location, we perform 1000 lookups in sequence and time the end-to-end completion time for each request. We use Amazon's official C# Software Development Kit (SDK) to access DynamoDB, with HTTP as the underlying transport protocol.

Our initial experiments uncovered a subtle parameter that had a very significant effect on performance. Each lookup is implemented as an HTTP POST request. Such requests might include a header field, "Expect: 100 continue", that instructs the server to reply with a 100 status code before the client will send the body of the POST request. This results in two rounds of communication for each request, since the client first sends the header of the request, and then waits for a reply before sending the body. Alternatively, clients can omit the "Expect: 100 continue" header field, and send the entire POST request as one TCP segment. In the common case, this means a POST request can be completed using just a single round of communication.

The default configuration of the .NET runtime does enable the "Expect: 100 continue" header field, which leads to surprising results. Since Jovaku sends lookups using the DNS protocol, which is again based on UDP, and only switches to HTTP for the final hop between the relay-node and the DynamoDB service, a Jovaku lookup only needs a single round trip of communication between the client and the cloud. So with the default .NET configuration, Jovaku outperforms DynamoDB even in the case when *all* lookups miss the cache. In short, requests take an apparent detour by going through DNS, but since they only travel most of that route once instead of twice, the longer route ends up being faster.

This finding, while interesting, is somewhat tangential to what we set out to investigate. The DynamoDB service responds to POST requests both with and without the "Expect: 100 continue" header, so performance-aware clients can reconfigure .NET to omit it. In the interest of fairness, we adopt this as our baseline. Another potential pitfall for naive clients is to set up new TCP connections for each HTTP request. This is unnecessary, as the same TCP connection can be reused for multiple HTTP requests, and this is what HTTP libraries such as *curl* and .NET generally do. To account for this, we exclude the first request from our measurements in this and all subsequent experiments. With these optimizations, a lookup will in the common case require only a single round-trip of communication.

Table 6.3 summarizes our baseline performance measurements, correlating them with the previously listed ping times. The effect of the "Expect: 100

Table 6.3: Average baseline lookup performance for the DynamoDB service in Ireland.

Label	With 100 continue	Without 100 continue	Ping Latency
Norway	135 ms	69 ms	64 ms
Norway-3G	355 ms	159 ms	116 ms
Norway-4G	261 ms	134 ms	120 ms
US West	368 ms	166 ms	155 ms
Asia	693 ms	353 ms	339 ms
Australia	712 ms	368 ms	365 ms

continue" header field is evident: when including the field, the average time to perform a lookup is about twice the time needed for a ping request, indicating that each request usually requires two round-trips of network communication. Without the field, the baseline lookup performance closely approaches that of a ping request, indicating that a single round-trip is sufficient. As noted, we assume that clients concerned with latency will optimize their configurations for that, so we adopt the latter measurements as the proper baseline performance for DynamoDB.

The numbers reported in Table 6.3 are averages of 1000 requests. Request times do not appear to follow a normal distribution; rather, the main bulk of requests are close to the minimum, while there are some rare outliers that take longer. We illustrate the distribution of request times by plotting histograms that show the percentage of requests that fall within each range of completion times. Figure 6.2 shows these distributions for all our client locations.

6.2.2 Jovaku Performance

In our next set of experiments, we measure lookup times when going through the Jovaku SDK. In this scenario, lookups are translated into DNS requests that are sent to the local DNS server. Depending on whether or not the value is cached, the DNS server either replies immediately, or forwards the request to the cloud, as illustrated in Figure 3.1. As noted, we measure the cache hit performance by warming up the DNS cache with long-lived entries, before measuring a sequence of lookups. We measure the cache miss performance by looking up values with TTL zero, which are never cached by DNS servers.

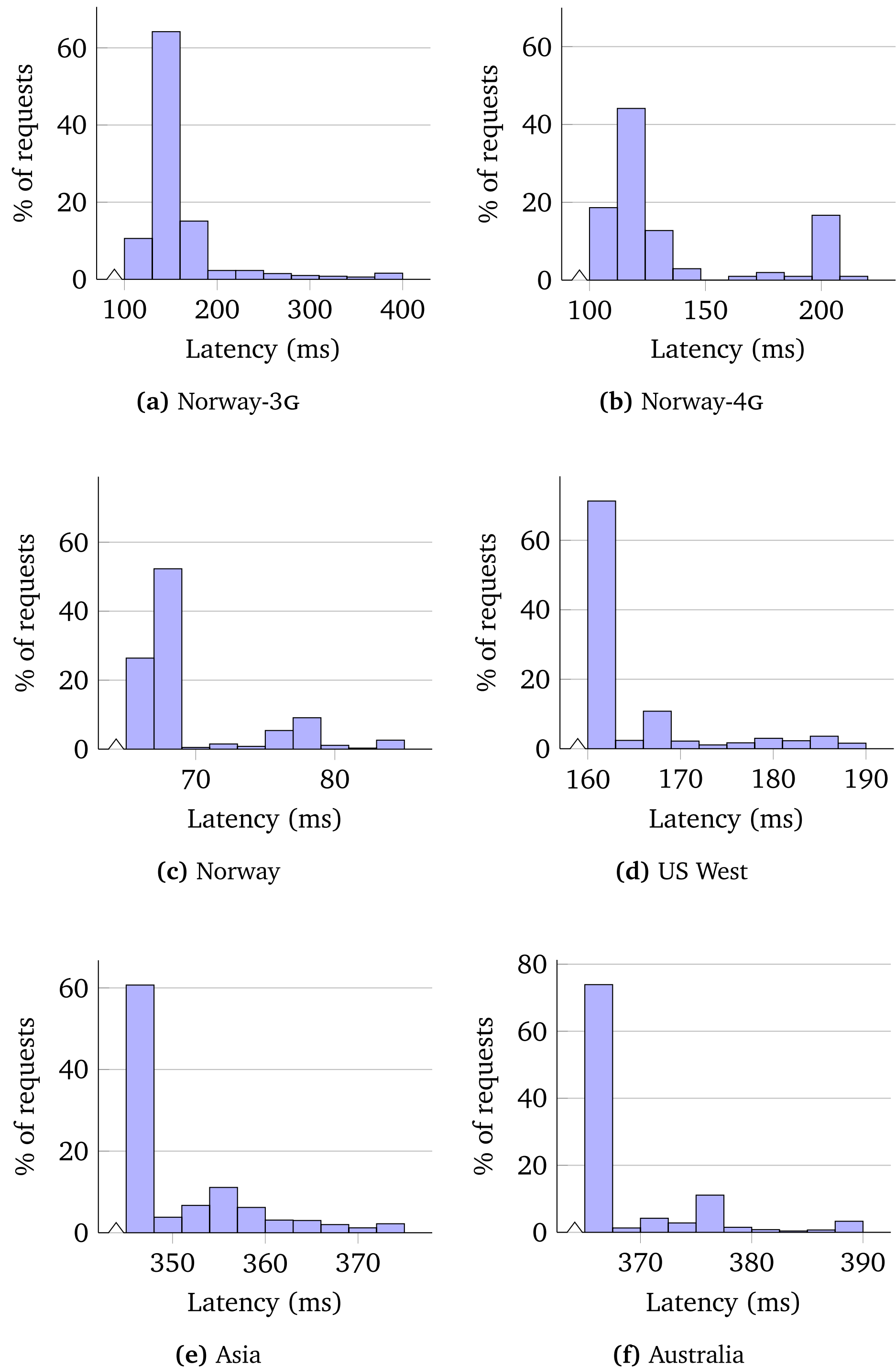


Figure 6.2: Distribution of baseline lookup performance for the DynamoDB service in Ireland using the official Amazon C# SDK.

Table 6.4: Average lookup performance using Jovaku with the DynamoDB service in Ireland.

Label	Baseline	Hit	Miss	Overhead	Break-even
Norway	69 ms	< 1 ms	78 ms	13.0 %	12.0 %
Norway-3G	159 ms	86 ms	168 ms	5.7 %	11.0 %
Norway-4G	134 ms	52 ms	183 ms	36.0 %	37.0 %
US West	166 ms	< 1 ms	186 ms	12.0 %	10.0 %
Asia	353 ms	< 1 ms	376 ms	6.5 %	6.1 %
Australia	368 ms	< 1 ms	385 ms	4.6 %	4.4 %

Table 6.4 shows the results. For each location, the table lists the baseline lookup performance, along with the measured latencies for cache hits and cache misses. As before, the reported numbers are averages of 1000 requests executed in sequence. We also calculate two additional metrics, listed in the last two columns.

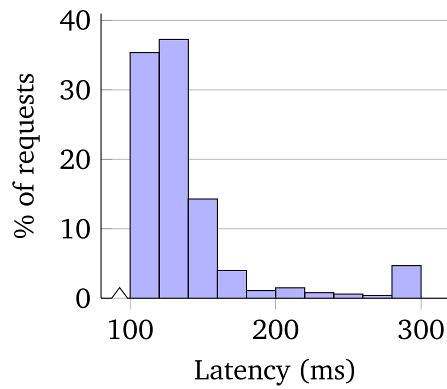
The **Overhead** column shows the relative overhead of a cache miss compared to the baseline lookup time. In the worst case, where every single request misses the cache, this is the resulting overhead of using Jovaku.

The **Break-even** column shows how many percent of lookup requests must hit the cache in order to break even in terms of average request times. This threshold relates to how the cache hit and cache miss times compare to the baseline performance. For example, if cache hits take half the time of the baseline, while cache misses take twice as long, 67 % of requests must hit the cache in order to break even.

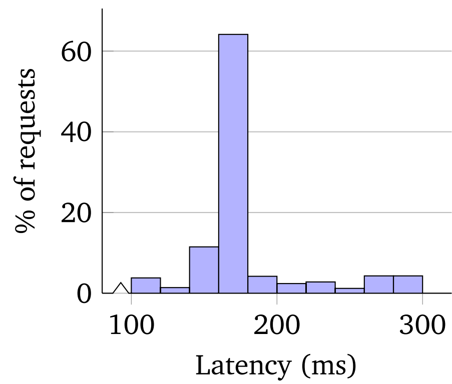
Given the average times for a cache hit T_{hit}, a cache miss T_{miss}, and the baseline latency T_{baseline} of a direct lookup, we define the break-even threshold as the fraction x where:

$$x \cdot T_{\text{hit}} + (1 - x) \cdot T_{\text{miss}} = T_{\text{baseline}}$$

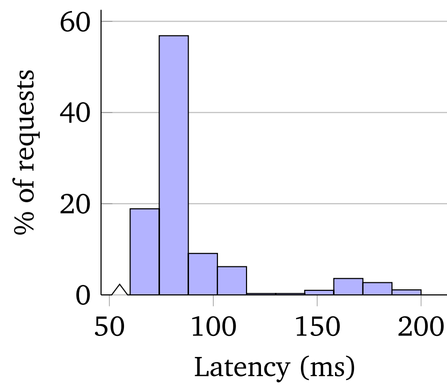
Or, equivalently:



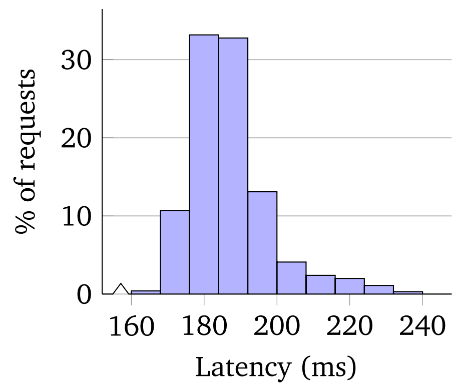
(a) Norway-3G, 0% cache hits



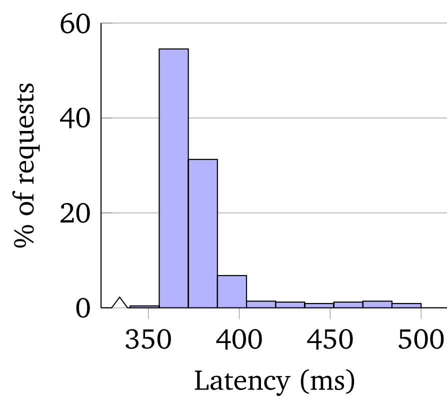
(b) Norway-4G, 0% cache hits



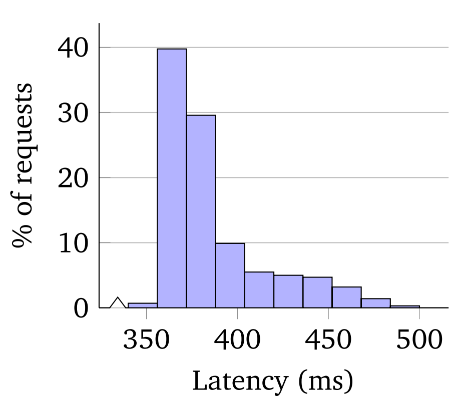
(c) Norway, 0% cache hits



(d) US West, 0% cache hits



(e) Asia, 0% cache hits



(f) Australia, 0% cache hits

Figure 6.3: Distribution of worst-case lookup performance with Jovaku, where no results are cached in local DNS servers.

$$x = \frac{(T_{\text{miss}} - T_{\text{baseline}})}{(T_{\text{miss}} - T_{\text{hit}})}$$

Intuitively, this means that x is a tipping point. When a fraction x of requests hit the cache, the average latency will be the same as without any caching. For higher ratios, the average latency will be better than the baseline, and for lower ratios, it will be worse.

Our results show that the overhead of a Jovaku cache miss is in the range of 4 % to 12 %. Figure 6.3 shows the distribution of request times for cache misses. We do not include the distribution for cache hits, since they are extremely fast for the majority of locations. This is because our EC2 nodes are located in data centers with high speed networks, and these nodes have very low latency access to their local DNS service.

The mobile networks in Norway are exceptions, with slower access to DNS. On the 3G network, the overhead of a cache miss is less than 6 %. This means that caching remains very justifiable, since a hit rate of 11 % would be sufficient to break even. The 4G network has higher overhead for cache misses, evidently taking a longer detour when going through DNS. This can be a side effect of additional hops with WiFi tethering, or poor 4G coverage in Tromsø at the time of the experiments. For mobile devices in more distant locations from Ireland, the relative difference between a cache hit and the baseline performance would likely be much greater, making caching even more attractive.

6.2.3 Jovaku with Alternative DNS Configuration

Given the fast DNS access times for the EC2 nodes in our previous experiments, it is natural to further explore the performance of Jovaku with alternative DNS configurations, i.e. when DNS access is more costly. We use EC2 nodes as a convenient way to experiment with different geographical locations, but mobile devices cannot reasonably be expected to have as fast access to DNS as computers in a data center. We therefore run additional experiments using Google's Public DNS service, creating more realistic access times for DNS lookups.

The setup remains the same as in the previous experiment, with the only difference being that DNS requests are sent to Google's DNS service instead of the local DNS service. All of our clients configure the anycast address `8.8.8.8` as their local DNS server, and requests are routed to the geographically closest Google DNS server through anycast routing.

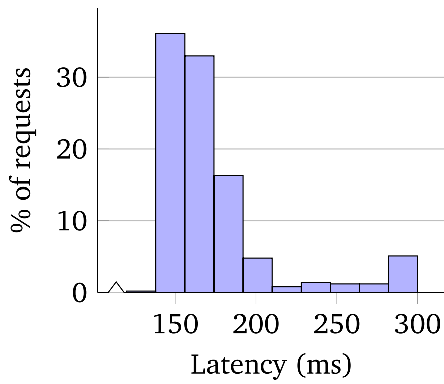
Table 6.5: Average lookup performance using Jovaku with the DynamoDB service in Ireland and Google Public DNS.

Label	Baseline	Hit	Miss	Overhead	Break-even
Norway	69 ms	60 ms	110 ms	59.0 %	82.0 %
Norway-3G	159 ms	106 ms	182 ms	14.0 %	30.0 %
Norway-4G	134 ms	75 ms	242 ms	80.0 %	65.0 %
US West	166 ms	25 ms	229 ms	38.0 %	31.0 %
Asia	353 ms	14 ms	371 ms	5.1 %	5.0 %
Australia	368 ms	3 ms	442 ms	20.0 %	17.0 %

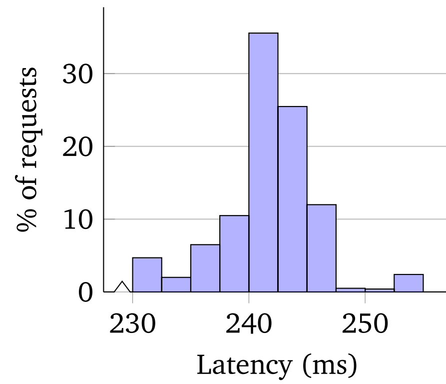
Table 6.5 summarizes the results with this configuration. As expected, the DNS configuration has a very direct effect on cache hit times, since these are directly determined by the latency to the DNS cache. For our clients in US West, Asia, and Australia, DNS access times remain modest after switching to Google's service. In Norway, Google's DNS service has a much higher latency, likely due to poor coverage. This includes the mobile clients, which also are located in Norway.

A more subtle result is how the DNS configuration impacts cache misses. Requests that miss the cache are forwarded to the relay-node in the cloud. This can also change the routing path of the request. For example, our client in Australia still enjoys very low-latency access to Google Public DNS, and yet its average latency for a cache miss is much worse when using this DNS service. On the other hand, while the mobile devices in Norway have relatively high latency for cache hits, the latency for a cache miss on the 3G network is only 14 % higher than the baseline, indicating that the actual detour of going through DNS is relatively short. As a result, caching remains beneficial in this scenario for hit rates of 30 % and higher. Figure 6.4 shows the distribution of request times for cache misses for all of the locations using this DNS configuration.

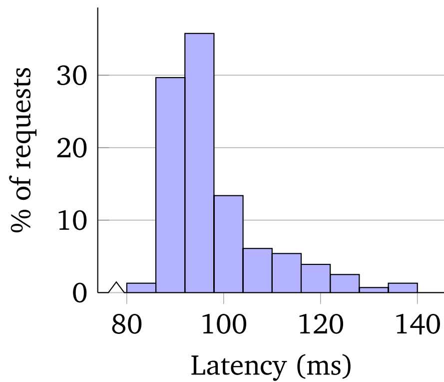
Overall, our evaluation shows that many variables affect the performance of Jovaku, and how beneficial caching will be in any given scenario. One general observation is that locations far away from the cloud database service, will benefit more from caching. This is to be expected, since remote locations have a higher baseline latency and can benefit relatively more from a cache hit.



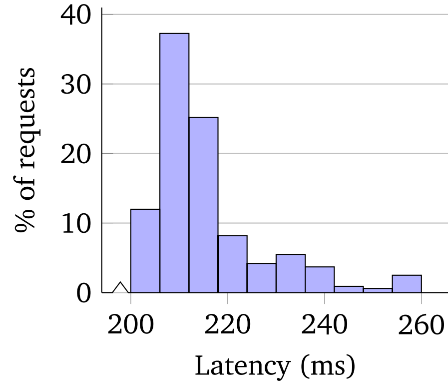
(a) Norway-3G, 0% cache hits



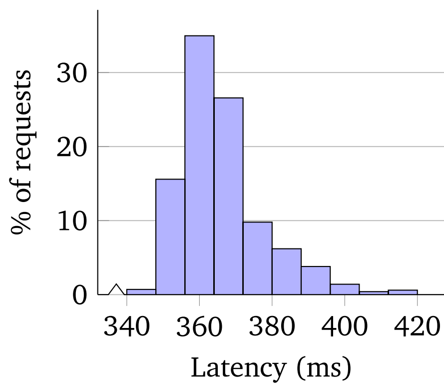
(b) Norway-4G, 0% cache hits



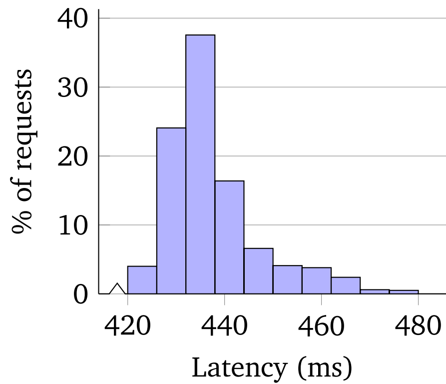
(c) Norway, 0% cache hits



(d) US West, 0% cache hits



(e) Asia, 0% cache hits



(f) Australia, 0% cache hits

Figure 6.4: Distribution of worst-case lookup performance with Jovaku and Google Public DNS, where no results are cached in Google's DNS servers.

Another important factor is the latency to the local DNS service, which determines the cost of a cache hit. In isolation, lower latency to the cache is always better. However, this does not necessarily imply that data should be cached as close to the clients as possible. The rate of cache hits might in some applications depend on how many clients share the same cache, so the ideal placement of caching nodes is a much more complex problem, in practice.

Finally, the worst-case overhead of a workload where all requests miss the cache is very important. If this overhead is low, caching will be justifiable for low hit rates, even if cache hits are relatively costly. Interactive applications may also be more concerned with bounding the worst case latency than with improving the average latency, as long as it stays within ranges that are imperceptible to users. For these applications, Jovaku is attractive for its ability to reduce the cost of operating the cloud database service, provided the worst-case latency has a reasonable bound.

Of the combinations we have experimented with, caching proved most beneficial for the clients in Asia and Australia. Due to the high baseline latency of these locations, the overhead of a cache miss is relatively low. Meanwhile, the savings in latency for cache hits are largest here, leading to break-even thresholds as low as 4.4 %. Norway-4G has the worst combination of circumstances when configured to use Google Public DNS. It has the lowest baseline latency to access DynamoDB in Ireland, combined with the highest latency to access Google Public DNS, which results in a break-even threshold as high as 82 %. However, using the local DNS service, the break-even threshold is only 12 %. These results gives an affirmative answer to Question 1 of the evaluation, and we can conclude that DNS is a viable placement for a caching middle tier.

Predicting how a client in any given location might perform would be impossible, even with much more extensive experiments. But our results do give some indications, and show that great variations must be expected. One pragmatic way to address these variations is to make clients adaptive; the baseline latency of directly accessing the database can be compared on the fly to the latency of going through DNS. Clients that observe unacceptable delays can try alternative DNS services or simply revert to the baseline performance. This kind of adaptive behavior could feasibly be built into the Jovaku client library, although we have not done so currently.

6.3 Black box testing

To establish the usefulness of satellite execution in Jovaku, we report on a black-box examination of the cloud communication patterns of some popular

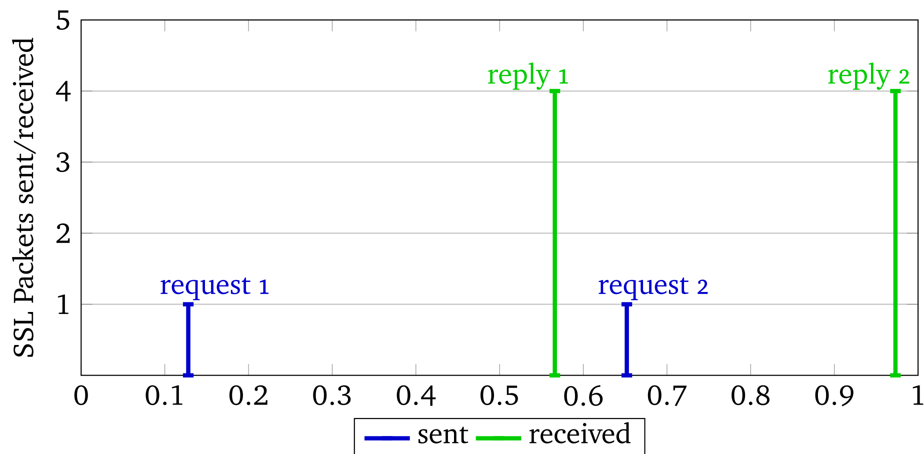


Figure 6.5: Example communication pattern between mobile device and cloud assumed to be of a request/reply type.

mobile/cloud applications. Here we seek to discover patterns consistent with sequences of dependent requests, with the motivation that satellite execution can be used in place of such interactions. We configure our mobile device to communicate through an access point instrumented to capture all ingress and egress network packets. We then inspect the encrypted TCP streams and dissect them into SSL packets, looking for what appears to be consecutive request/reply cloud interactions without intervening user actions. The particular pattern we looked for is exemplified in Figure 6.5, which shows two interactions assumed to be of a request/reply type.

Our findings for cloud interactions during startup of four popular applications are summarized in Figure 6.6. Connections to advertisement networks have been filtered out, to focus on connections that are related to the functional operation of the respective application.

We observed that the applications communicate over a number of separate network connections, ranging from 2 for the social networking application to 12 for the short messaging application. Most of these connections are to different services within the same cloud, but some are external, typically in support of content distribution such as Akamai [126].

The number of assumed request/reply interactions varied across applications and connections, with the instant- and short messaging applications respectively having as many as 7 and 6 consecutive interactions. These findings suggest satellite execution can be effective if applied in these popular applications, and gives an affirmative answer to Question 2 of the evaluation.

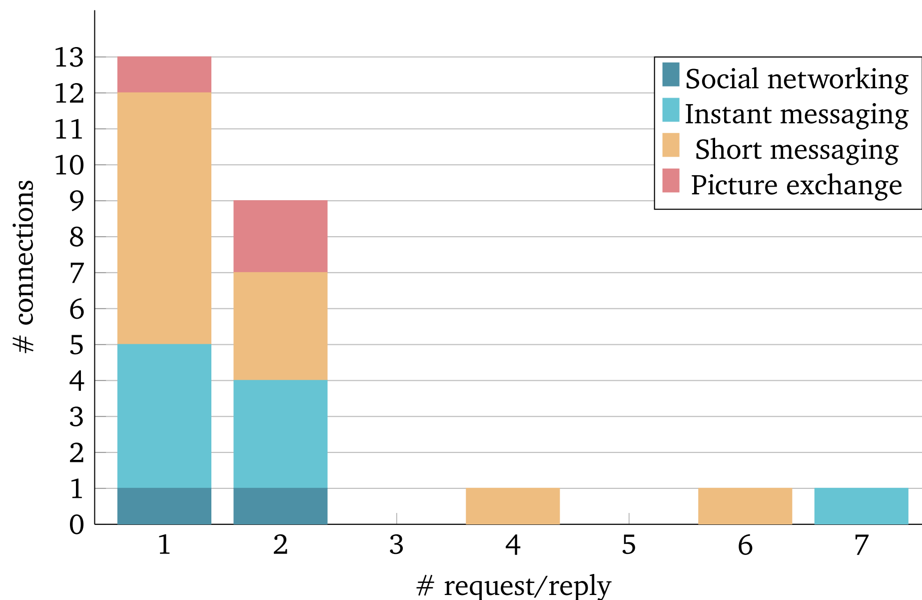


Figure 6.6: Summary of cloud interactions during various mobile application startup.

6.4 Satellite execution

To establish whether satellite execution can reduce communication latency for mobile/cloud applications, we have to ascertain communication overhead is minimized, and quantify communication latency when a client issues cloud database queries directly and when using satellite execution.

We start by measuring the resulting byte arrays from the custom serialization algorithm listed in Code Listing 4.12, with a varying number of queries. We then compare it to the byte arrays produced with the default serialization algorithm. The results can be seen in Figure 6.7. The custom serialization algorithm produces byte arrays of less than half the size compared to the default algorithm, when considering 1 to 10 queries. We also observe that the default algorithm grows at a slower rate than the custom algorithm. Mobile functions containing a large number of queries, might consider using the default algorithm, or split the mobile function into several smaller mobile functions, to obtain a byte array with the least number of bytes.

To quantify communication latency, we use the bag-of-queries implementation outlined in Code Listing 4.11 to issue queries to the DynamoDB instance located in Ireland. We use the desktop and mobile platform located in Tromsø to issue requests, and a EC2 t1.micro instance to host the relay-node located in the same availability zone as the DynamoDB instance. The locations are illustrated in Figure 6.8.

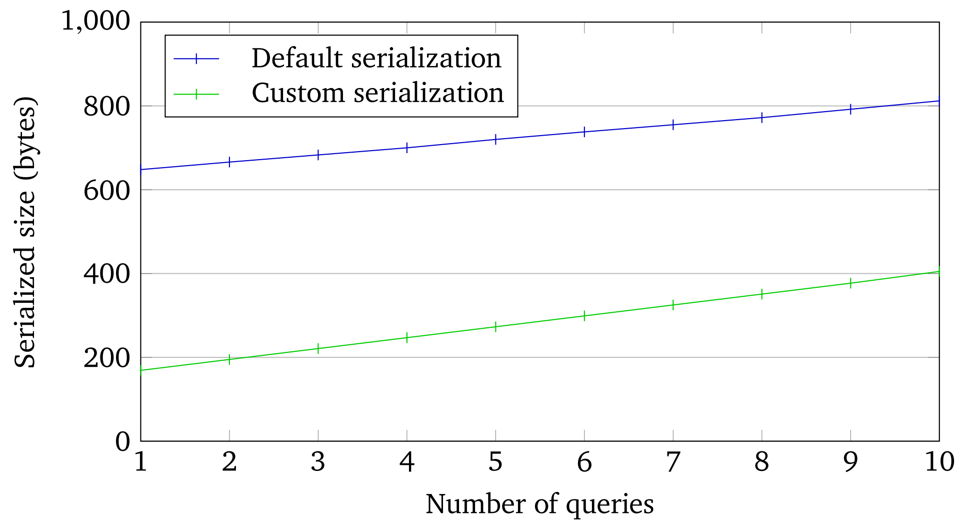


Figure 6.7: Comparison of the default serialization and the custom serialization algorithms, with respect to the size of the resulting byte array.

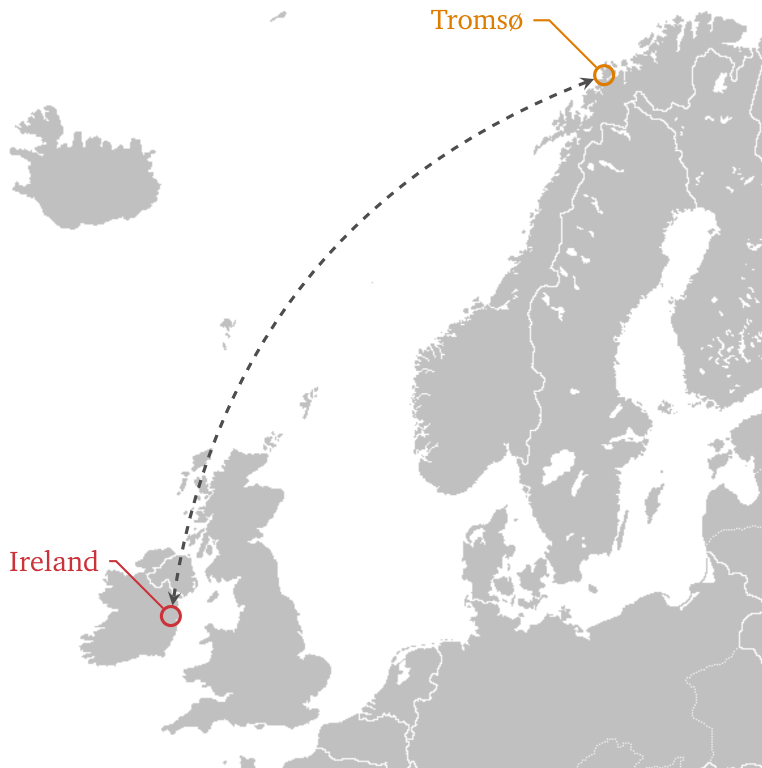


Figure 6.8: Locations of nodes involved in the experiment.

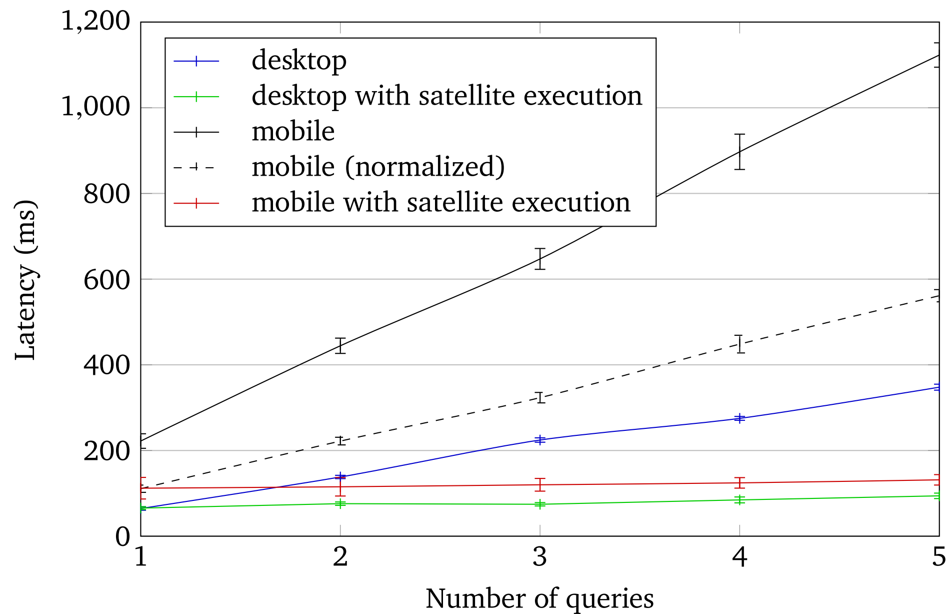


Figure 6.9: Observed mean latency when executing a varying number of cloud database queries with and without satellite execution. The error bars show the standard deviation.

Latency when the bag contained between 1 and 5 queries is shown in Figure 6.9. Results are the mean latency over 1000 runs. As shown, there are significant latency savings when the bag contains more than one query. This is because latency between the relay-node and the database is low, and the round-trip latency between the client and the cloud—approximately 64 ms for desktop and 105 ms for mobile—overshadows the low cost of serializing and transferring the query bag.

The DynamoDB library uses the HTTP 100-continue feature when interacting with the cloud database. Use of this feature adds a communication round-trip to database interaction, needlessly inflating latency [58]. We therefore used platform interfaces to disable this HTTP feature on desktop. Similar interfaces do not exist on Windows Phone, however. The results in Figure 6.9 consequently include one additional round-trip latency for mobile, compared to desktop. To better convey the latency difference between mobile and desktop, the figure also includes results where one round-trip latency has been subtracted from mobile. Even after this normalization, mobile has significantly higher latency than desktop, demonstrating the relative importance of our satellite execution technique for the mobile platform.

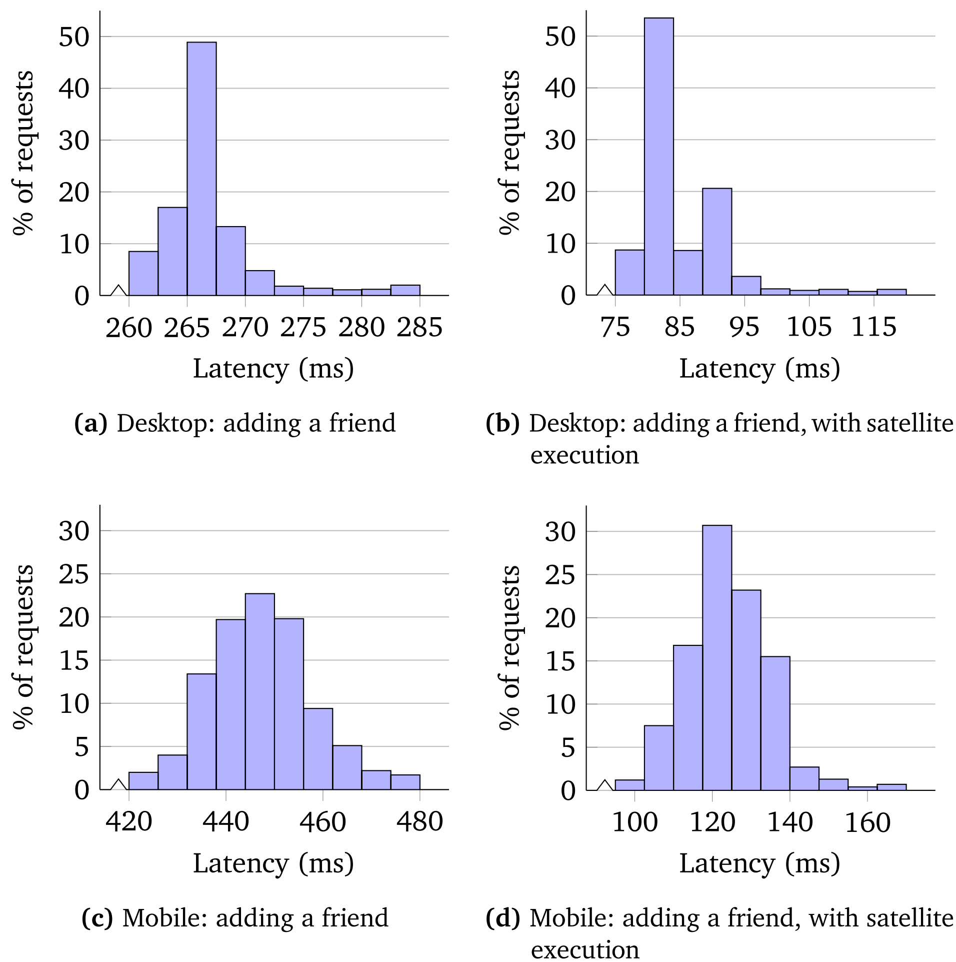


Figure 6.10: Distribution of latencies when adding a friend to a social network, with and without satellite execution.

The data on popular applications in Figure 6.6 only indicates that latency savings are possible; determining the degree to which the interaction could exploit satellite execution would require access to application source code.

To approximate the savings that could be experienced in a deployed application we reconstruct a scenario where a friend connection is established in the MSRBook, a social networking application based on Deuteronomy [127]. The addition of a friend in this network involves a friend and news feed update for both concerned parties, for a total of 4 queries. Equivalent queries were placed in our bag-of-queries and we run the friend-add action 1000 times on both the desktop and the mobile platform, with and without satellite execution. Figure 6.10 illustrates latency savings. Savings due to satellite execution are pronounced; on desktop latency drops from a mean of 266 ms to 85 ms, while

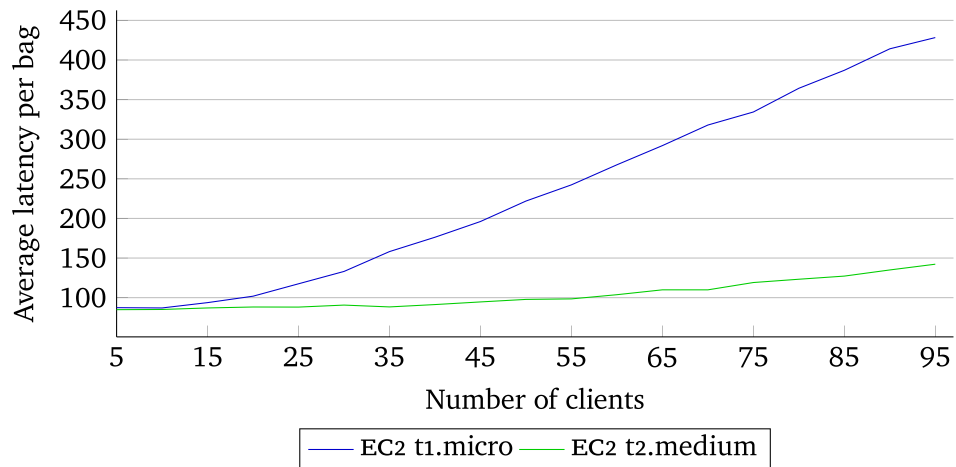


Figure 6.11: Latency per bag-of-queries when increasing the number of clients that concurrently submit mobile functions to a relay-node.

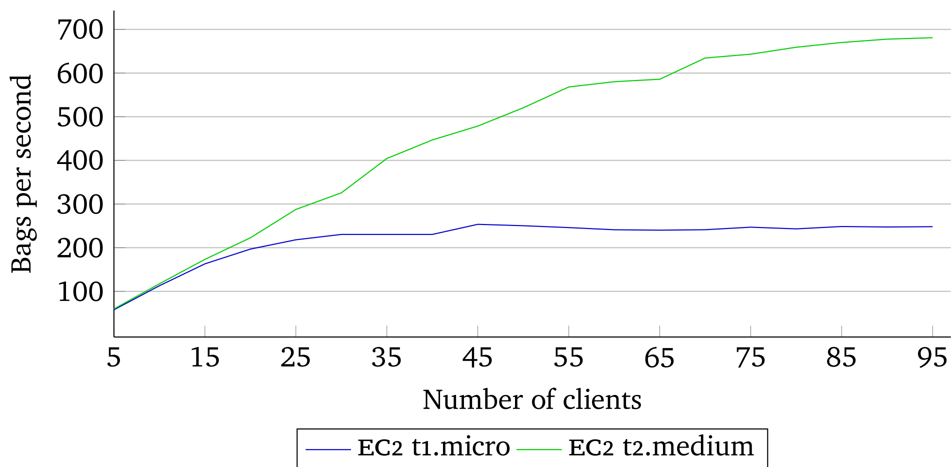
it drops on mobile from a mean of 448 ms to 124 ms.

A latency reduction of 68 % and 72 % respectively, gives an affirmative answer to Question 3 of the evaluation, proving the effectiveness of satellite execution for reducing latency. With affirmative answers to Questions 1, 2 and 3, we can evaluate the relay-node to answer Question 4.

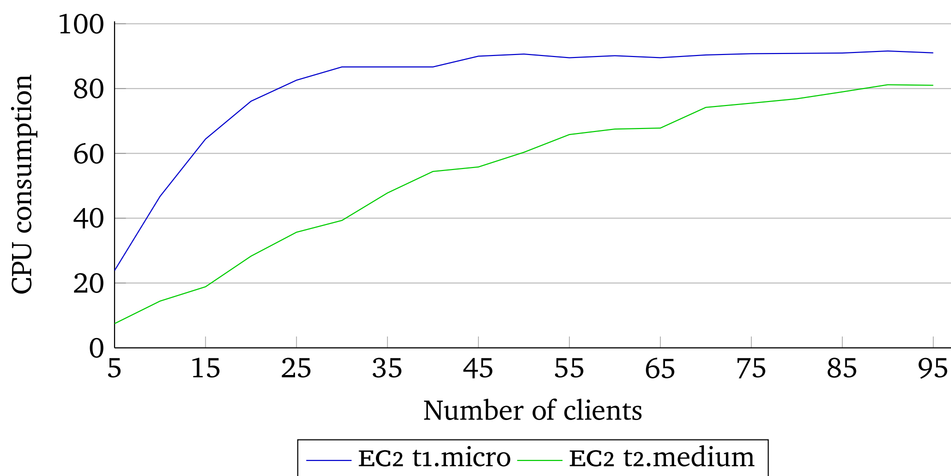
6.5 Relay-node performance

On a mobile device such as a smartphone, a person uses around 24 different applications every month [66]. Even the modest resource allocations available to the Amazon t1.micro instance used in our experiments are likely to be ample for a relay-node dedicated to a single mobile device. But if the relay-node functionality was a service offered by the cloud database provider, in a fashion similar to the Parse platform [86], the relay-node would likely be shared among many mobile devices and its capacity would be an issue. We therefore last consider an experiment where the relay-node serves an increasing number of mobile devices.

In the experiment, we configured a thread on the desktop machine in Norway to repeatedly submit mobile functions to the relay-node, in a closed loop. Each mobile function was a bag of 4 queries. We then increased the number of threads, simulating an increase in clients, to ensure high contention for relay-node resources, in an attempt to reveal the capacity for executing mobile



(a) Average throughput as the number of clients increase.



(b) Average CPU consumption as the number of clients increase.

Figure 6.12: Average CPU consumption and throughput at the relay-node when increasing the number of concurrent clients that submit mobile functions.

functions.

We repeated the experiment both for the EC2 t1.micro and t2.medium instances. Results are shown in Figures 6.11 and 6.12. In Figure 6.12a we observe that the t1.micro instance is capable of completing around 250 bags per second before throughput levels off. The t2.medium instance peaks at around 700 bags per second. As the number of clients continues to increase, each of them observes higher latency, as illustrated in Figure 6.11. In Figure 6.12b, we observe

a close correlation between throughput and Central Processing Unit (CPU) consumption for both instance types. This indicates that CPU is the likely bottleneck that causes throughput to peak.

The experiment does not expose any scalability issues in our relay-node implementation, with regard to concurrently serving an increasing number of clients. Throughput levels off and remains stable after it peaks. A single relay-node can thus be shared among multiple mobile devices, and also across different applications. These results give an answer to Question 4 of the evaluation, which seeks to evaluate how the relay-node scales as the workload increases.

6.6 Summary

Our evaluation involves clients from around the world, on different continents, accessing an Amazon DynamoDB database hosted in Ireland. We also vary the DNS configuration, using Google Public DNS as an alternative service. Across configurations, the experiments show some variations in the cost of handling cache hits and misses. In general, the benefits of using Jovaku for a given client depends on its proximity to the DynamoDB service, and its DNS access latency. Overall, caching is beneficial in the great majority of configurations. Clients will generally benefit from Jovaku even with cache hit rates as low as 5 % to 10 %.

To estimate the potential for satellite execution in real applications, our evaluation examines the communication patterns of some popular applications through a black-box technique. This has yielded indications that dependent queries occur in practice, since sequences of up to 7 requests were observed back-to-back over the same connection on startup. Looking at a concrete implementation of a social networking application from [127], we found specific examples. For example, a friend request results in 4 dependent queries; when offloaded to the cloud from a phone using satellite execution, the completion time of a friend request dropped from 450 ms to approximately 125 ms, reducing latency by more than 72 %.

The experimental results give affirmative answers to all the questions in our evaluation, and support our thesis that it is possible to leverage existing infrastructure to provide a generic middle tier to reduce latency for mobile/cloud applications.

7 Concluding Remarks

We conclude this dissertation by summarizing and restating our findings, focusing on how they corroborate and affirm our thesis. Based on this, we draw three main conclusions. Finally, we highlight avenues for improvement and how these limitations could be addressed in future work.

To recapitulate our starting point for this work, the thesis of this dissertation is that:

A generic middle tier can leverage existing infrastructure to reduce latency for mobile/cloud applications.

To evaluate this thesis, we created the Jovaku system. Jovaku consists of a middle tier component and a Software Development Kit (SDK). The middle tier is positioned strategically between the mobile device and the cloud, providing lower latency access to data stored in the cloud. The SDK allows developers of mobile/cloud applications to make use of existing Domain Name System (DNS) infrastructure to cache database values close to the mobile device, and offload code from the application to the cloud for execution on the middle tier component, with low latency access to other cloud resources.

In Chapter 3 we present the Jovaku architecture that allows developers to exploit existing DNS infrastructure to cache cloud database values. As the evaluation in Section 6.2 shows, this architecture proves efficient to reduce communication latency for applications reading from cloud databases, even

with hit rates as low as 4.4 %.

In Chapter 4 we present the satellite execution extension to Jovaku. Satellite execution allows portions of code in mobile/cloud applications to be offloaded to a relay-node in the cloud, where access to other cloud resources can be accessed with lower latency. As the evaluation in Section 6.4 shows, this approach can reduce latency for some applications by as much as 72 % on mobile devices.

The addition of a middle tier like Jovaku has interesting implications for mobile/-cloud application architectures. The desire to reduce latency for mobile/cloud applications traditionally tends to encourage a split application architecture. Parts of the application logic execute on the mobile device, and other parts execute in the cloud. The split between device and cloud is application-specific and the operations exposed by the backend are tailored to avoid extraneous communication rounds.

Many frameworks and platforms aim to ease the development of mobile/cloud applications that are factored into separate device and cloud components. The cloud component typically provides a backend-as-a-service solution that offers backend cloud storage, as well as the ability to deploy application modules that execute in the cloud. The implication is that processing occurs close to the data. One common downside of this approach is that the device-specific and cloud-specific parts of the application are deployed independently, through different channels. This increases the risk of Application Programming Interface (API) version incompatibility, when old versions deployed on devices interact with newer version deployed in the cloud.

With Jovaku, the ability to do caching in the middle tier reduces the demand for custom backend APIs, as generic cloud database services become more viable as out-of-the-box backends. Meanwhile, the satellite execution capabilities offer a way to build modular applications where some code may execute in the cloud, without mandating two separate deployment channels. In Chapter 5 we present Picster and Dapper, two modern mobile/cloud proof-of-applicability prototypes that make use of Jovaku. The applications have been tested in real deployment, with positive results. This shows that Jovaku is viable beyond laboratory experimentation, and can be used for constructing modern mobile/cloud applications.

7.1 Conclusions

Based on the work presented in this dissertation, we draw the following three conclusions:

1. Jovaku allows developers to exploit existing DNS infrastructure to reduce communication latency for mobile/cloud applications when communicating with a cloud database.

2. Jovaku allows developers to dynamically offload code from mobile/cloud applications to the cloud to reduce latency for interacting with cloud services.

3. Jovaku is viable beyond laboratory experimentation, and can be used to create modern mobile/cloud applications.

Hence, we conclude that *a generic middle tier can leverage existing infrastructure to reduce latency for mobile/cloud applications*, and our main thesis holds.

7.2 Future Work

Our experiments in Section 6.2 revealed interesting performance variations between the local DNS server, external DNS servers, and using a direct database connection. Choosing the optimal configuration when dealing with mobile devices is not straightforward. Variables like mobile coverage, cache hit rate, and routing distance to the cloud database will invariably change as the mobile device moves. One way to address this issue is to extend the Jovaku client library with adaptive behavior. As the mobile device moves, instrumentation of observed latency could form the basis for choosing the optimal DNS server, or for choosing to revert back to a direct connection.

As mobile/cloud applications grow more complex, mobile functions will also tend to grow more complex. Mobile functions can make use of several third-party libraries to extend the basic functionality, such as encryption, social network, or cloud storage libraries. As Chapter 4 describes, the responsibility of making such libraries available to the relay-node lies on the client, which may create additional round-trips of communication between the client and relay-node.

These libraries are often published to trusted package management systems,

such as NuGet[1], and are digitally signed. As such, the relay-node can be extended with means to resolve these dependencies dynamically and retrieve them from the package management system instead.

In our most recent work, we have built LADY [61], a system that augments the .NET platform with a highly reliable mechanism for resolving and loading assemblies and arranges for safe execution of partially trusted code. This system can be used as an improvement in Jovaku to further reduce communication latency, by decoupling the mechanism for assembly resolution from the protocol for offloading mobile functions. Evaluation shows that LADY can dramatically reduce the latency incurred when missing assemblies must be resolved at the relay-node.

1. NuGet is a package management system, closely integrated with Microsoft Visual Studio.

Bibliography

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," Tech. Rep. EECS-2009-28, UC Berkeley Reliable Adaptive Distributed Systems Laboratory, 2009.

[2] I. Zhang, A. Szekeres, D. V. Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy, "Customizable and extensible deployment for mobile/cloud applications," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, (Broomfield, CO), pp. 97–112, USENIX Association, 10 2014.

[3] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches," *Wireless Communications and Mobile Computing*, vol. 13, no. 18, pp. 1587–1611, 2013.

[4] X. Jin and Y.-K. Kwok, "Cloud assisted p2p media streaming for bandwidth constrained mobile subscribers," in *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pp. 800–805, IEEE, 2010.

[5] James Hamilton, "The cost of latency." `http://perspectives.mvdirona.com/2009/10/the-cost-of-latency/`.

[6] Jake Brutlag, "Speed Matters." `http://googleresearch.blogspot.no/2009/06/speed-matters.html`.

[7] E. Schurman and J. Brutlag, "The user and business impact of server delays, additional bytes, and http chunking in web search," in *Velocity Web Performance and Operations Conference*, 2009.

[8] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber, *Comparative evaluation of latency reducing and tolerating techniques*, vol. 19. ACM, 1991.

[9] T. M. Kroeger, D. D. Long, and J. C. Mogul, "Exploring the bounds of web latency reduction from caching and prefetching.," in *USENIX Symposium on Internet Technologies and Systems*, pp. 13–22, 1997.

[10] L. Fan, P. Cao, W. Lin, and Q. Jacobson, "Web prefetching between low-bandwidth clients and proxies: potential and performance," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, pp. 178–187, ACM, 1999.

[11] N. Benton, L. Cardelli, and C. Fournet, "Modern concurrency abstractions for c#," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 26, no. 5, pp. 769–804, 2004.

[12] R. Hu, N. Yoshida, and K. Honda, "Session-based distributed programming in java," in *ECOOP 2008–Object-Oriented Programming*, pp. 516–541, Springer, 2008.

[13] J. Diaz, C. Munoz-Caro, and A. Nino, "A survey of parallel programming models and tools in the multi and many-core era," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, no. 8, pp. 1369–1386, 2012.

[14] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan, "Timecard: Controlling user-perceived delays in server-based mobile applications," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 85–100, ACM, 2013.

[15] R. Escriva, B. Wong, and E. G. Sirer, "Hyperdex: a distributed, searchable key-value store," in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '12, (New York, NY, USA), pp. 25–36, ACM, 2012.

[16] J. Pokorny, "Nosql databases: a step to database scalability in web environment," *International Journal of Web Information Systems*, vol. 9, no. 1, pp. 69–82, 2013.

[17] A. B. M. Moniruzzaman and S. A. Hossain, "Nosql database: New era of databases for big data analytics - classification, characteristics and comparison," *CoRR*, vol. abs/1307.0191, 2013.

[18] R. Hecht and S. Jablonski, "Nosql evaluation: A use case oriented survey," 2011.

[19] K. Kaur and R. Rani, "Modeling and querying data in nosql databases," in *Big Data, 2013 IEEE International Conference on*, pp. 1–7, IEEE, 2013.

[20] S. Lombardo, E. Di Nitto, and D. Ardagna, "Issues in handling complex data structures with nosql databases," in *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*, pp. 443–448, IEEE, 2012.

[21] M. Satyanarayanan, "Cloudlets: at the leading edge of cloud-mobile convergence," in *Proceedings of the 9th international ACM SIGSOFT conference on Quality of software architectures*, pp. 1–2, ACM, 2013.

[22] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "Comet: code offload by migrating execution transparently," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 93–106, USENIX Association, 2012.

[23] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the 6th conference on Computer systems*, EuroSys '11, (New York, NY, USA), pp. 301–314, ACM, 2011.

[24] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, (New York, NY, USA), pp. 49–62, ACM, 2010.

[25] E. Tilevich and Y.-W. Kwon, "Cloud-based execution to improve mobile application energy efficiency," *Computer*, vol. 47, pp. 75–77, Jan. 2014.

[26] G. Peng, "CDN: content distribution network," *CoRR*, vol. cs.NI/0411069, 2004.

[27] F. B. Schneider, "Byzantine generals in action: implementing fail-stop processors," *ACM Transactions on Computer Systems*, vol. 2, no. 2, pp. 145–154, 1984.

[28] P. J. Denning, "Is computer science science?," *Communications of the ACM*, vol. 48, no. 4, pp. 27–31, 2005.

[29] D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, P. R. Young, and P. J. Denning, "Computing as a discipline," *Communications of the ACM*, vol. 32, no. 1, pp. 9–23, 1989.

[30] D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young, "Computing as a discipline," *Communications of the ACM*, vol. 32, no. 1, pp. 9–23, ACM, 1989.

[31] D. Parnas and P. Clements, "A rational design process: How and why to fake it," *Software Engineering, IEEE Transactions on*, vol. SE-12, pp. 251–257, 2 1986.

[32] D. E. Avison and J. Pries-Heje, *Research in information systems: A handbook for research supervisors and their students*. Gulf Professional Publishing, 2005.

[33] A. Kvalnes, D. Johansen, R. van Renesse, F. Schneider, and S. Valvag, "Omni-kernel: An operating system architecture for pervasive monitoring and scheduling," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, pp. 2849–2862, Oct 2015.

[34] Å. Kvalnes, D. Johansen, R. v. Renesse, and A. Audun, "Vortex: an event-driven multiprocessor operating system supporting performance isolation," tech. rep., University of Tromsø, 2003.

[35] A. Kvalnes, R. v. Renesse, and D. Johansen, "Performance isolation and adaption in the vortex kernel," tech. rep., Universitetet i Tromsø, 2003.

[36] A. Nordal, Å. Kvalnes, and D. Johansen, "Paravirtualizing tcp," in *Proceedings of the 6th international workshop on Virtualization Technologies in Distributed Computing Date*, pp. 3–10, ACM, 2012.

[37] A. Ø. Nordal, Å. Kvalnes, R. Pettersen, and D. Johansen, "Streaming as a hypervisor service," in *Proceedings of the 7th international workshop on Virtualization technologies in distributed computing*, pp. 33–40, ACM, 2013.

[38] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of the 6th symposium on Operating Systems Design and Implementation*, OSDI '04, pp. 137–150, USENIX Association, 2004.

[39] D. Johansen, K. J. Lauvset, R. van Renesse, F. B. Schneider, N. P. Sudmann, and K. Jacobsen, "A TACOMA retrospective," *Software - Practice and Experience*, vol. 32, pp. 605–619, 2001.

[40] D. Johansen, K. Marzullo, and K. J. Lauvset, "An approach towards an agent computing environment," in *ICDCS'99 Workshop on Middleware*,

1999.

[41] D. Johansen, “Mobile agents: Right concept, wrong approach,” in *In 5th IEEE International Conference on Mobile Data Management (MDM 2004*, pp. 300–301, IEEE Computer Society, 2004.

[42] S. V. Valvåg and D. Johansen, “Oivos: Simple and efficient distributed data processing,” in *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications*, HPCC ’08, pp. 113–122, IEEE Computer Society, 2008.

[43] S. V. Valvåg and D. Johansen, “Update maps - a new abstraction for high-throughput batch processing,” in *Proceedings of the 2009 IEEE International Conference on Networking, Architecture, and Storage*, NAS ’09, pp. 431–438, IEEE Computer Society, 2009.

[44] S. V. Valvåg, D. Johansen, and A. Kvalnes, “Cogset: A high performance MapReduce engine,” *Concurrency and Computation: Practice and Experience*, vol. 25, no. 1, pp. 2–23, 2013.

[45] S. V. Valvåg, D. Johansen, and A. Kvalnes, “Position paper: Elastic processing and storage at the edge of the cloud,” in *Proceedings of the 2013 International Workshop on Hot Topics in Cloud Services*, HotTopiCS ’13, (New York, NY, USA), pp. 43–50, ACM, 2013.

[46] A. Nordal, A. Kvalnes, J. Hurley, and D. Johansen, “Balava: Federating private and public clouds,” in *Services (SERVICES), 2011 IEEE World Congress on*, pp. 569 –577, july 2011.

[47] D. Johansen and J. Hurley, “Overlay cloud networking through meta-code,” in *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual*, pp. 273–278, IEEE, 2011.

[48] H. Johansen, A. Allavena, and R. Van Renesse, “Fireflies: scalable support for intrusion-tolerant network overlays,” in *ACM SIGOPS Operating Systems Review*, vol. 40, pp. 3–13, ACM, 2006.

[49] H. D. Johansen, R. V. Renesse, Y. Vigfusson, and D. Johansen, “Fireflies: a secure and scalable membership and gossip service,” *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 2, p. 5, 2015.

[50] H. Johansen, D. Johansen, and R. van Renesse, “Firepatch: secure and time-critical dissemination of software patches,” in *Proceedings of the 22nd IFIP International Information Security Conference*, pp. 373–384,

IFIP, May 2007.

[51] H. K. Stensland, V. R. Gaddam, M. Tennøe, E. Helgedagsrud, M. Næss, H. K. Alstad, A. Mortensen, R. Langseth, S. Ljødal, O. Landsverk, C. Griwodz, P. Halvorsen, M. Stenhaug, and D. Johansen, "Bagadus: An integrated real-time system for soccer analytics," *To appear in ACM Transactions on Multimedia Computing, Communications and Applications*, 2014.

[52] P. Halvorsen, S. Sægrov, A. Mortensen, D. K. C. Kristensen, A. Eichhorn, M. Stenhaug, S. Dahl, H. K. Stensland, V. R. Gaddam, C. Griwodz, and D. Johansen, "Bagadus: An integrated system for arena sports analytics - a soccer case study.," in *Proceedings of the 4th ACM International Conference on Multimedia Systems (MMSys)*, (Oslo, Norway), ACM, To appear March 2013.

[53] D. Johansen, M. Stenhaug, R. Hansen, A. Christensen, and P.-M. Hogmo, "Muithu: Smaller footprint, potentially larger imprint," in *Digital Information Management (ICDIM), 2012 Seventh International Conference on*, pp. 205 –214, aug. 2012.

[54] M. Stenhaug, Y. Yang, C. Gurrin, and D. Johansen, "Muithu: A touch-based annotation interface for activity logging in the norwegian premier league," in *MultiMedia Modeling*, pp. 365–368, Springer, 2014.

[55] R. van Renesse, H. Johansen, N. Naigaonkar, and D. Johansen, "Secure abstraction with code capabilities," *arXiv preprint arXiv:1210.5443*, 2012.

[56] D. Johansen, H. Johansen, T. Aarflot, J. Hurley, Å. Kvalnes, C. Gurrin, S. Zav, B. Olstad, E. Aaberg, T. Endestad, *et al.*, "Davvi: A prototype for the next generation multimedia entertainment platform," in *Proceedings of the 17th ACM international conference on Multimedia*, pp. 989–990, ACM, 2009.

[57] D. Johansen, P. Halvorsen, H. Johansen, H. Riiser, C. Gurrin, B. Olstad, C. Griwodz, Å. Kvalnes, J. Hurley, and T. Kupka, "Search-based composition, streaming and playback of video archive content," *Multimedia Tools and Applications*, vol. 61, no. 2, pp. 419–445, 2012.

[58] R. Pettersen, S. V. Valvåg, A. Kvalnes, and D. Johansen, "Jovaku: Globally distributed caching for cloud database services using DNS," in *IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, pp. 127–135, 2014.

[59] R. Pettersen, S. V. Valvåg, A. Kvalnes, and D. Johansen, "Cloud-side execution of database queries for mobile applications," in *CLOSER 2015 : Proceedings of the 5th International Conference on Cloud Computing and Services Science*, pp. 586–594, 2015.

[60] R. Pettersen, S. V. Valvåg, Å. Kvalnes, and D. Johansen, "Using satellite execution to reduce latency for mobile/cloud applications," in *Cloud Computing and Services Science*, vol. 581, pp. 279–298, Springer, 2015.

[61] S. V. Valvåg, R. Pettersen, H. Johansen, and D. Johansen, "Lady: Dynamic resolution of assemblies for extensible and distributed .net applications," in *CLOSER 2016 : Proceedings of the 6th International Conference on Cloud Computing and Services Science*, To be presented in 2016.

[62] Wikipedia, "HTC Dream." https://en.wikipedia.org/wiki/HTC_Dream.

[63] Forbes, "Microsoft's Upcoming Lumia 950 and 950 XL Windows 10 Flagship Smartphones Pictured In Latest Leaks." http://www.forbes.com/sites/marcochiappetta/2015/08/28/microsofts-upcoming-lumia-950-and-950-xl-windows-10-flagship-smartphones-pictured-in-latest-leaks/.

[64] "Android Anatomy and Physiology." http://androidteam.googlecode.com/files/Anatomy-Physiology-of-an-Android.pdf.

[65] H. T. Al-Rayes, "Studying main differences between android & linux operating systems," *International Journal of Electrical & Computer Sciences IJECS-IJENS*, vol. 12, no. 05, 2012.

[66] Nielsen, "Smartphones: So many apps, so much time." http://www.nielsen.com/us/en/insights/news/2014/smartphones-so-many-apps--so-much-time.html.

[67] "ARM TrustZone." http://www.arm.com/products/processors/technologies/trustzone/.

[68] "Intel Software Guard Extensions (SGX)." https://software.intel.com/en-us/isa-extensions/intel-sgx.

[69] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using arm trustzone to build a trusted language runtime for mobile applications," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 67–80, ACM, 2014.

[70] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, (Broomfield, CO), pp. 267–283, USENIX Association, Oct. 2014.

[71] H.-S. Oh, B.-J. Kim, H.-K. Choi, and S.-M. Moon, "Evaluation of android dalvik virtual machine," in *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, (New York, NY, USA), pp. 115–124, ACM, 2012.

[72] "VM Ware." `http://www.vmware.com`.

[73] "Oracle Virtual Box." `http://www.virtualbox.org`.

[74] "Docker." `http://www.docker.com`.

[75] A. Inc., "Swift. A modern programming language that is safe, fast, and interactive.." `https://developer.apple.com/swift/`.

[76] Anandtech, "A Closer Look at Android RunTime (ART) in Android L." `http://anandtech.com/show/8231/a-closer-look-at-android-runtime-art-in-android-l`.

[77] "Android 5.0 Lollipop, thoroughly reviewed." `http://arstechnica.com/gadgets/2014/11/android-5-0-lollipop-thoroughly-reviewed`.

[78] Deepti Prakash, "Compile in the Cloud with WP8." `http://blogs.msdn.com/b/msgulfcommunity/archive/2013/03/16/compile-in-the-cloud-with-wp8.aspx`.

[79] D. Hermes, "Mobile development using xamarin," in *Xamarin Mobile Application Development*, pp. 1–8, Springer, 2015.

[80] "Xamarin." `http://www.xamarin.com`.

[81] Statistic Brain, "Youtube Statistics," 2015. `http://www.statisticbrain.com/youtube-statistics/`.

[82] Statistic Brain, "Facebook Statistics," 2015. `http://www.statisticbrain.com/facebook-statistics/`.

[83] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin, "Incoop: Mapreduce for incremental computations," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, p. 7, ACM, 2011.

[84] J. Dean and S. Ghemawat, “MapReduce: A flexible data processing tool,” *Communications of the ACM*, vol. 53, no. 1, pp. 72–77, ACM, 2010.

[85] R. T. Fielding and R. N. Taylor, “Principled design of the modern web architecture,” in *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE ’00, (New York, NY, USA), pp. 407–416, ACM, 2000.

[86] Parse. `http://www.parse.com`.

[87] “Google Cloud Platform.” `https://cloud.google.com/`.

[88] “Amazon Web Services.” `https://aws.amazon.com/`.

[89] B. Leiba, “Oauth web authorization protocol,” *IEEE Internet Computing*, vol. 16, no. 1, p. 74, 2012.

[90] “Memcached.” `http://www.memcached.org/`.

[91] “Redis.” `http://redis.io`.

[92] G. Pierre and M. van Steen, “Globule: a collaborative content delivery network,” *Communications Magazine*, vol. 44, no. 8, pp. 127–133, 2006.

[93] “Akamai Technologies.” `http://www.akamai.com/html/technology/index.html`.

[94] “Limelight Networks.” `http://www.limelightnetworks.com/platform/cdn/`.

[95] “Rapid Edge Content Delivery Network.” `http://www.rapidedgecdn.com/`.

[96] “Amazon CloudFront.” `http://aws.amazon.com/cloudfront/`.

[97] P. Mockapetris and K. J. Dunlap, *Development of the domain name system*, vol. 18. ACM, 1988.

[98] Spotify LABS, “In praise of boring technology.” `https://labs.spotify.com/2013/02/25/in-praise-of-boring-technology/`.

[99] David Leadbeater, “Wikipedia over DNS,” 2009. `https://dgl.cx/wikipedia-dns`.

[100] X. Chen, H. Wang, S. Ren, and X. Zhang, "Maintaining strong cache consistency for the domain name system," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 19, pp. 1057–1071, Aug 2007.

[101] S. Sivasubramanian, "Amazon dynamodb: a seamlessly scalable non-relational database service," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 729–730, ACM, 2012.

[102] "ISC Bind." `https://www.isc.org/downloads/bind/`.

[103] "Bind DLZ." `http://bind-dlz.sourceforge.net/`.

[104] "The Amazon DynamoDB API." `http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/API.html`.

[105] "libCurl library." `http://curl.haxx.se/libcurl/`.

[106] "Jansson JSON Library." `http://www.digip.org/jansson/`.

[107] "BeeCrypt Cryptography Library." `http://sourceforge.net/projects/beecrypt/`.

[108] "Norid DNS Checker." `http://dnscheck.norid.no/`.

[109] A. Li, X. Yang, S. Kandula, and M. Zhang, "CloudCmp: comparing public cloud providers," in *ACM SIGCOMM*, pp. 1–14, 2010.

[110] I. Goldberg, D. Wagner, R. Thomas, E. A. Brewer, *et al.*, "A secure environment for untrusted helper applications: Confining the wily hacker," in *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography*, vol. 6, pp. 1–1, 1996.

[111] A. Fuggetta, G. Picco, and G. Vigna, "Understanding code mobility," *Software Engineering, IEEE Transactions on*, vol. 24, pp. 342–361, 5 1998.

[112] "Wireshark network protocol analyzer." `https://www.wireshark.org/`.

[113] Microsoft, "Application Domains," 2015. `http://msdn.microsoft.com/en-us/library/cxk374d9%28v=vs.90%29.aspx`.

[114] J. W. Stamos and D. K. Gifford, "Remote evaluation," *ACM Transactions on Programming Languages and Systems*, vol. 12, pp. 537–564, 10 ACM,

1990.

[115] J. W. Stamos, “Remote evaluation.,” tech. rep., DTIC Document, 1986.

[116] E. Zayas, “Attacking the process migration bottleneck,” *ACM SIGOPS Operating Systems Review*, vol. 21, no. 5, pp. 13–24, 1987.

[117] T. McDonnell, B. Ray, and M. Kim, “An empirical study of api stability and adoption in the android ecosystem,” in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pp. 70–79, Sept 2013.

[118] “Yelp.” `http://www.yelp.com`.

[119] “Around Me.” `http://www.aroundme.com`.

[120] “iGroups: Apple’s New iPhone Social App in Development.” `http://www.patentlyapple.com/patently-apple/2010/03/igroups-apples-new-iphone-social-app-in-development.html`.

[121] M. Rogowsky, “Why don’t cell phones work at music festivals?.” `https://www.quora.com/Why-dont-cell-phones-work-at-music-festivals`.

[122] N. Cohen, “Hong kong protests propel firechat phone-to-phone app,” *The New York Times*, 2014.

[123] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, “Epidemic algorithms for replicated database maintenance,” in *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pp. 1–12, ACM, 1987.

[124] T. McDonnell, B. Ray, and M. Kim, “An empirical study of api stability and adoption in the android ecosystem,” in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pp. 70–79, IEEE, 2013.

[125] J.-J. Dubray, “Understanding the Cost of Versioning an API.” `http://www.ebpml.org/blog2/index.php/2013/11/25/understanding-the-costs-of-versioning`.

[126] E. Nygren, R. K. Sitaraman, and J. Sun, “The akamai network: A platform for high-performance internet applications,” *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 2–19, Aug. 2010.

[127] J. J. Levandoski, D. B. Lomet, M. F. Mokbel, and K. Zhao, “Deuteronomy: Transaction support for cloud data.,” in *CIDR*, pp. 123–133,

www.cidrdb.org, 2011.

Publications

This dissertation is based on the work presented in the following four publications:

Publication I

R. Pettersen, S. V. Valvåg, A. Kvalnes, and D. Johansen, "Jovaku: Globally distributed caching for cloud database services using DNS", in *IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, pp. 127—135, 2014.

In this paper we present Jovaku, a generic caching layer for cloud database services that can induce significant performance improvements and cost savings. Jovaku demonstrates the viability of a truly global caching infrastructure by building on the existing DNS infrastructure. Database operations are relayed through the DNS protocol, allowing results to be cached in DNS servers close to client devices. This greatly simplifies deployment, and offers supreme availability, allowing devices anywhere to benefit from database caching. The evaluation shows that the latency to access Amazon DynamoDB is significantly reduced for requests that hit the cache, and that applications can benefit from caching with hit rates as low as 5 %. The system is described in detail in Chapter 3.

Publication II and III

R. Pettersen, S. V. Valvåg, A. Kvalnes, and D. Johansen, "Cloud-side execution of database queries for mobile applications", in *CLOSER 2015 : Proceedings of the 5th International Conference on Cloud Computing and Services Science*, pp. 586-–594, 2015.

In this paper we demonstrate a practical way to reduce latency for mobile .NET applications that interact with cloud database services. We provide a programming abstraction for location-independent code, which has the potential to execute either locally or at a satellite execution environment in the cloud, in close proximity to the database service. This preserves a programmatic style of database access, and maintains a simple deployment model, but allows applications to offload latency-sensitive code to the cloud. The evaluation shows that this approach can significantly improve the response time for applications that execute dependent queries, and that the required cloud-side resources are modest. The system is described in detail in Chapter 4.

In addition to being accepted at the conference with a 15 % acceptance rate, the paper appeared on a short list of papers nominated for best paper award. The paper was also among 14 papers of the 146 accepted papers selected for revised publishing in Springer's Cloud Computing and Services Science.

R. Pettersen, S. V. Valvåg, Å. Kvalnes, and D. Johansen, "Using satellite execution to reduce latency for mobile/cloud applications", in *Cloud Computing and Services Science*, pp. 279—298, Springer, 2015.

Publication IV

S. V. Valvåg, R. Pettersen,H. Johansen, and D. Johansen, "Lady: Dynamic resolution of assemblies for extensible and distributed .net applications", in *CLOSER 2016 : Proceedings of the 6th International Conference on Cloud Computing and Services Science*, To be presented in Rome, April 23rd, 2016.

In this paper we describe LADY, a system that augments the .NET platform with a highly reliable mechanism for resolving and loading assemblies and arranges for safe execution of partially trusted code. This system can be used as an improvement in Jovaku to further reduce communication latency, by decoupling the mechanism for assembly resolution from the protocol for offloading mobile functions. Evaluation shows that LADY can dramatically reduce the latency incurred when missing assemblies must be resolved at the relay-node.

Other Publications

During the course of the study for this dissertation, the author has also contributed to the following publication, which are related but not part of this dissertation:

A. Ø. Nordal, Å. Kvalnes, R. Pettersen, and D. Johansen, "Streaming as a hypervisor service", in *Proceedings of the 7th international workshop on Virtualization technologies in distributed computing*, pp. 33–40, ACM, 2013.