Using Satellite Execution to Reduce Latency for Mobile/Cloud Applications

Robert Pettersen, Steffen Viken Valvåg,
Åge Kvalnes, and Dag Johansen

Department of Computer Science,
University of Tromsø,
The Arctic University of Norway
{robert,steffenv,aage,dag}@cs.uit.no
http://www.cs.uit.no

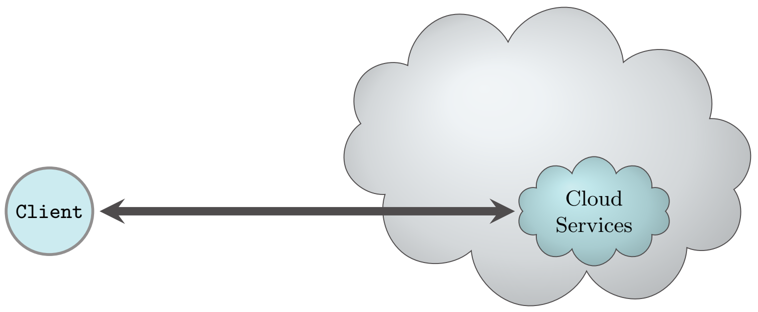
Abstract. We demonstrate a practical way to reduce latency for mobile .NET applications that interact with cloud services, without disrupting application architectures. We provide a programming abstraction for location-independent code, which has the potential to execute either locally or at a satellite execution environment in the cloud, where other cloud services can be accessed with low latency. This maintains a simple deployment model, but gives applications the option to offload latency-sensitive code to the cloud. Services like cloud databases can still be accessed programmatically, but with less concern for the aggregated latency of consecutively-issued requests. Our evaluation shows that this approach can significantly improve the response time for applications that execute dependent database queries, and that the required cloud-side resources are modest.

Keywords: Mobile, Cloud, Performance, Latency, Satellite Execution, Code Offloading, Cloud Databases

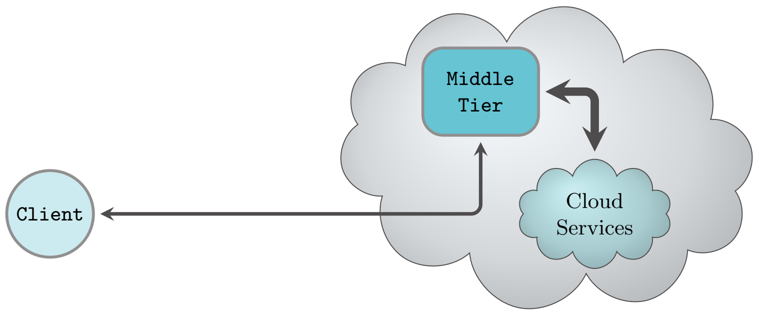
1 Introduction

Use of cloud-provided services is integral to the operation of modern distributed and mobile applications. For example, 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.



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



(b) With satellite execution.

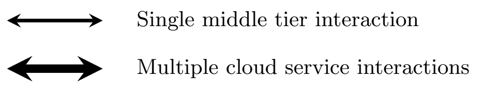

Single middle tier interaction
Multiple cloud service interactions

Fig. 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—reducing overall latency.

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, we have measured a latency of 50ms–350ms for accessing the Amazon DynamoDB [7] cloud database from a mobile device [18], whereas a study covering 260 global vantage points reports an average round-trip time (RTT) of 74ms for accessing Amazon EC2 instances [13]. Issuing a sequence of queries to the cloud can result in unwanted delays that are perceptible by users.

One way to alleviate this problem is to move the execution of queries to a middle tier that 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. If an application experiences high latency, or needs to issue a long sequence of database queries, the queries can be offloaded to the cloud and executed in close proximity to the database service.

In this paper, we refine and generalize this idea, to reduce latency for any mobile/cloud application that issues a sequence of dependent requests to the cloud. By moving the code that accesses cloud services to a middle tier, positioned in close proximity to the cloud, the code can execute in an environment with lower latency. We refer to this concept as *satellite execution*, and illustrate it in Figure 1. Figure 1a shows the baseline scenario, where a client must send multiple requests over a high latency mobile network to the cloud in order to complete a task. These can be replaced with a single round-trip as in Figure 1b, where code is moved to the middle tier before multiple requests with intracloud latency are issued to the cloud service.

By implementing general-purpose offloading of code, and not just specializing on relaying of database queries, we preserve the programmatic style of database access, and its associated advantages. For example, offloaded code can perform computations, transformations, cryptographic operations, and any other manipulations of parameters and intermediate results that may be required when performing a sequence of queries. The increased flexibility also widens the applicability of satellite execution as a general concept.

To illustrate the benefits of our approach, we quantify latency savings when cloud database queries are executed from the middle tier rather than at the client-side device. We also examine communication traces of popular phone applications to determine the practicality of our approach, to see if real applications exhibit access patterns that are conducive to latency savings through satellite execution.

We implement satellite execution in a system called Dapper, which significantly extends and integrates the functionality of two previous systems: Rusta [23] and Jovaku [18]. Rusta is a platform for developing cloud applications that can utilize client-side storage and processing capacity, while the Jovaku system provides a distributed infrastructure for caching of cloud database values through the ubiquitous DNS service.

A goal with Rusta was to express computations in a location-independent way, allowing for opportune execution in the cloud or at client-side devices. This was accomplished by expressing computations in the Scala programming language and using built-in closure features to create transferable execution contexts. In Dapper we take a similar approach, but target the .NET platform, so that code in any of the .NET languages can be made transferable. Jovaku's architecture includes a cloud-side relay-node that bridges the DNS protocol with the database API. Dapper extends this component to include a middle-tier platform for hosting and safely executing offloaded code.

The rest of this paper is structured as follows. Section 2 elaborates on the background and context of our work, motivating our general approach. Section 3 describes the design and implementation of Dapper, and its programming abstractions for satellite execution. Section 4 contains a performance evaluation that focuses on the cloud database use-case, with measurements of typical reductions in latency, and the maximum query processing throughput that can be achieved in various configurations. Section 5 discusses related work, and Section 6 concludes.

2 Background

The desire to reduce latency for mobile/cloud applications tends to encourage a split application architecture, where parts of the application logic executes on the device, and other parts execute in the cloud. Higher-level operations such as submitting a comment or generating a news feed are delegated as a whole to the cloud, to avoid multiple round-trips of communication.

The split between frontend and backend also has a tangential benefit: it allows a variety of frontends, often tailored for different devices, to access the same backend service. For example, an on-line chess service will typically offer both a web-based frontend, as well as clients for various mobile devices and platforms. Users should be able to switch seamlessly between client devices, e.g. moving from their laptop to their phone, so the state of on-going games must be maintained by the backend. This requires frequent communication with the cloud to retrieve and update application state.

Many frameworks and platforms aim to ease the development of mobile applications that are factored into separate backend and frontend components. One example is Parse [17], which 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, close to the data. One common downside of these approaches is that the device-specific and cloud-specific parts of the application are deployed independently, through different channels. This increases the risk of breakage, when old versions deployed on devices interact with the newest version deployed in the cloud.

We approach the problem differently; rather than explicitly deploying parts of applications in the cloud, we empower applications to offload latency-sensitive code on demand, in a dynamic manner. Offloaded code will execute in close proximity to the backend storage service, where latency is low. Thus, we address the main motivating concern—improving application responsiveness as experienced by users—without dictating a static deployment model for applications.

A key idea underlying this work is to move computations closer to the data that they touch, which is a well-known technique that finds diverse applications. When processing streams of data, the demand for network bandwidth can be reduced by filtering streams closer to the source, pushing computations upstream. When processing stored data, similar gains can be made by scheduling computations to execute locally on the storage nodes, using functional programming models like MapReduce [6] for location independence.

Our experience from mobile agents [10, 11, 9] and MapReduce-style distributed data processing have inspired some key aspects of this work. As in Cogset [21], 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 moves back and forth between a client device and the cloud, it retains the defining ability to carry state.

3 Dapper

Instead of statically partitioning mobile/cloud applications into client-side and cloud-side components, satellite execution enables individual objects to move dynamically between the client and the cloud. The decision to deploy an object for satellite execution is taken at run-time. Deployment to the cloud involves moving an object's code (i.e., its class) and its current state. Incurred state changes while executing remotely are included when the object is moved back to the client. Objects can move repeatedly between the client and the cloud, for example in response to changes in application environment or state.

Jovaku's application-transparent interfacing with cloud databases through DNS was in part made possible by a cloud-side relay-node. The relay-node bridges the DNS infrastructure with the underlying cloud database service by turning DNS requests into database queries. The relay-node was placed in close proximity to the cloud database service 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 Dapper was an intuitive solution. To realize the satellite execution concept, we therefore extended the relay-node with capabilities for hosting and executing offloaded .NET code. Two main components were identified as necessities for this extension:

An execution environment that is capable of hosting multiple securely isolated *sandboxes*. Each of these sandboxes must be capable of loading and executing code on behalf of clients, without interfering with each other, or compromising the integrity of the surrounding execution environment.

A message processor that will receive and process messages sent from clients and demultiplex and pass messages on to the execution environment. There are several feasible approaches to implementing an efficient message processor. The Windows Communication Foundation (WCF) offers one convenient framework, but we decided to use an asynchronous socket-based server with a customized communication protocol, because this gave slightly better throughput.

An overview of the extended architecture with the new components can be seen in Figure 2. The Name Server is a BIND [2] server with a custom DLZ [1] driver that resolves DNS queries by accessing a cloud database service. As noted, this functionality stems from the original Jovaku system and complements the satellite execution capabilities of Dapper.

In addition to these new relay-node components, we saw the need to create a programming abstraction for execution of offloaded code. To this end, we defined the `IMobileFunction` interface, seen in Code Listing 1, which clients use to specify offloadable code. Implementations of this interface are called *mobile*

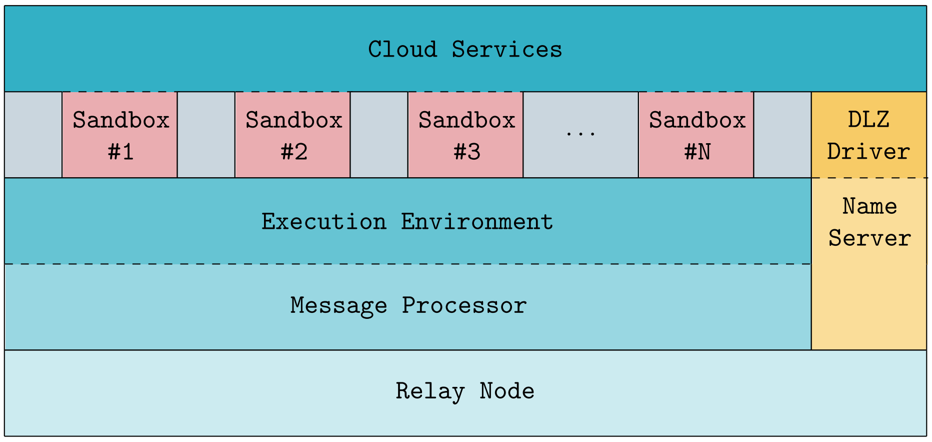


Fig. 2: An overview of the extended architecture where the message processor and execution environment have been integrated into the relay-node. The execution environment is capable of hosting multiple isolated sandboxes for loading and executing offloaded client code.

Code Listing 1: Interface to be implemented by mobile functions.

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

functions, as they can be serialized and moved for remote execution on a relay-node. The entry point of a mobile function is its `Execute` method, which may be invoked asynchronously using .NET's task-based asynchronous pattern.

Mobile functions contain user-defined code, and are black boxes to Dapper. Being implemented in C# or another .NET language, 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 safety. Dapper will only invoke 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, Dapper will let mobile functions access *safe* implementations of selected operations through the `IContext` interface, shown in Code Listing 2. These operations can involve I/O, but they are implemented by Dapper, with rigorous validation of arguments to minimize the potential for abuse.

Since we have selected cloud database services as a use case to focus on, our `IContext` interface provides basic key/value operations that can be supported

Code Listing 2: Interface for accessing cloud-side resources from a mobile function.

```
public interface IContext
{
    Task<object> Get(string key);

    Task<List<object>> GetMany(string key);

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

Code Listing 3: Interface for requesting remote execution of a mobile function.

```
public interface IDapper
{
    Task<object> ExecuteAt(IMobileFunction function,
        Uri location = null);
}
```

by any common NoSQL database. When applying satellite execution in other contexts, the interface would be extended correspondingly, to expose the relevant cloud service functionality.

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 fully client-side process by providing a context object that binds to a local database.

3.1 Implementation

Our current implementation targets Amazon's DynamoDB, which is a popular NoSQL cloud database service. The relay-node is manifested as an instance in the EC2 computing cluster, where DynamoDB can be accessed with very low network latency. In addition to the BIND process that serves DNS traffic, a separate server process—written in C#—implements the message processor and execution environment. Incoming messages 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 Dapper. Deserialization of a mobile function can fail, if its assembly is missing. In that

Code Listing 4: Creating a new application domain, with minimal permissions and a 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();
}
```

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.

From the client application's perspective, mobile functions are regular objects that may, upon request, be executed remotely. The `ExecuteAt` method in Code Listing 3 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.

In the relay-node, we sandbox the execution of mobile functions using .NET application domains [3], 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

Code Listing 5: 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 Task<string> Get(string key) { ... }

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

    [SocketPermission(Assert, Unrestricted = true)]
    [ReflectionPermission(Assert, Unrestricted = true)]
    [WebPermission(Assert, Unrestricted = true)]
    public Task<bool> Put(string key, object value) { ... }
}
```

and isolate execution of code. Dapper 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 the file system, or participating in bot-nets that deplete network resources. Code Listing 4 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 Dapper's own assemblies, and the official DynamoDB API from Amazon.

When mobile functions execute, they can use the `IContext` interface to access cloud services. Dapper implements this interface in the `SecureContext` class shown in Code Listing 5. The various database operations that can be performed are implemented using Amazon's API, which requires certain additional permissions to work correctly. Socket and web permissions are needed to create sockets and sending web requests, and the API 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

Code Listing 6: 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) { ... }
}
```

can be elevated selectively by marking the relevant methods with special security attributes. So 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 6 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, without issue.

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

Code Listing 7: Implementation of a bag-of-queries abstraction as a mobile function that can execute remotely in the cloud via satellite execution.

```
[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.GetMany(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;
    }
}
```

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`

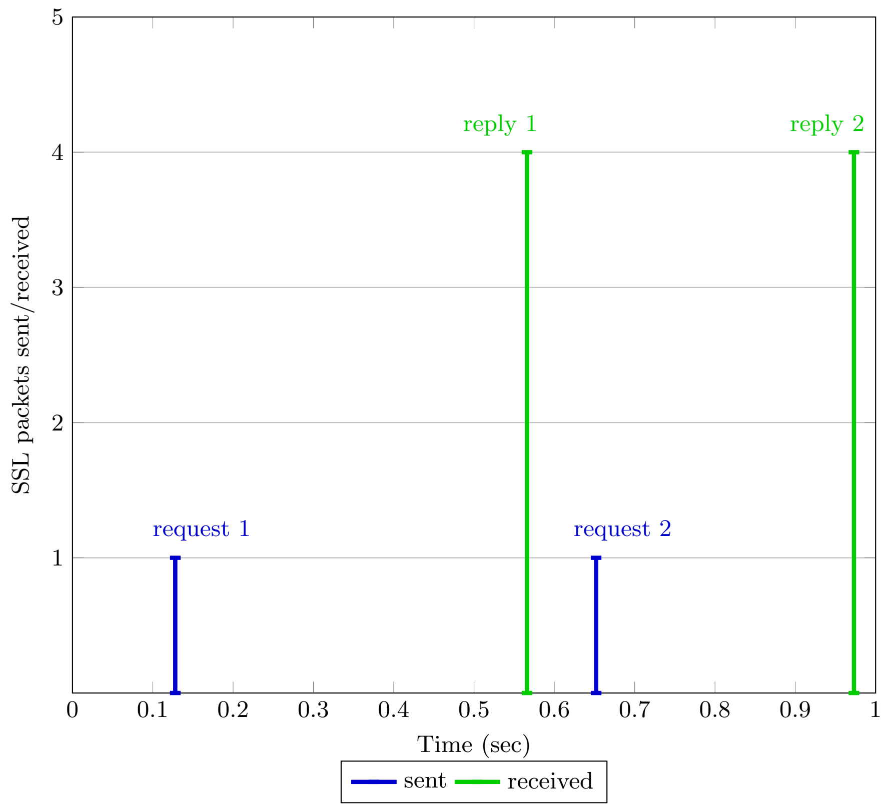


Fig. 3: Example communication pattern between mobile device and cloud assumed to be of a request/reply type.

method completes, the mobile function is again serialized into a byte array, using `SerializeFunction`, and passed back to the client.

Code Listing 7 shows an example of a mobile function that implements a bag-of-queries abstraction. Database queries are added to the bag by invoking `AddQuery`; the queries are aggregated in the `_queryList` field. The `Execute` method issues the aggregated queries via the context object and stores results in the `_responseList` field. After executing the mobile function, the client can observe the results of the queries by invoking `GetResponses`.

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.

Table 1: Summary of cloud interactions during phone application startup.

Application	**# request/reply**	**# connections**
Social networking	1	1
	2	1
Instant messaging	1	4
	2	3
	7	1
Short messaging	1	7
	2	3
	4	1
	6	1
Picture exchange	1	1
	2	2

4 Evaluation

Dapper runs on a variety of Microsoft Windows platforms, including phone, store, and desktop. We used two different client-side platforms in our experiments: (1) a phone with 2 GB memory and a quad-core QualComm Snapdragon 800 2.2 GHz CPU and (2) a desktop machine with 64 GB memory and a quad-core Intel Xeon E5-1620 3.7 GHz CPU. The phone ran Windows Phone 8.1 and communicated over 4G, whereas the desktop machine ran Windows 10 and was connected to a LAN.

The relay-node was hosted on two types of Amazon EC2 instances. The first type was t1.micro, equipped with 613 MB memory and a single-core 64-bit vCPU operating at 1.85 GHz. The second type was t2.medium, equipped with 4 GB memory and a dual-core vCPU operating at 2.50 GHz. Both types of instances were running Microsoft Windows Server 2012 R2. We used Amazon's DynamoDB as the cloud-side database, instantiated in the same availability zone as our relay-node.

We first report on a black-box examination of the cloud communication patterns of some popular mobile applications. Here we sought to discover patterns consistent with sequences of dependent requests, with the motivation that satellite execution could be used in place of such interactions. We configured our phone platform to communicate through an access point instrumented to capture all ingress and egress network packets. We then inspected the encrypted TCP streams and dissected them into SSL packets, looking for what appeared as consecutive request/reply cloud interactions without intervening user actions. The particular pattern we looked for is exemplified in Figure 3, 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 Table 1. We observed that the applications communicate over a number of separate network connections, ranging from 2 for the social

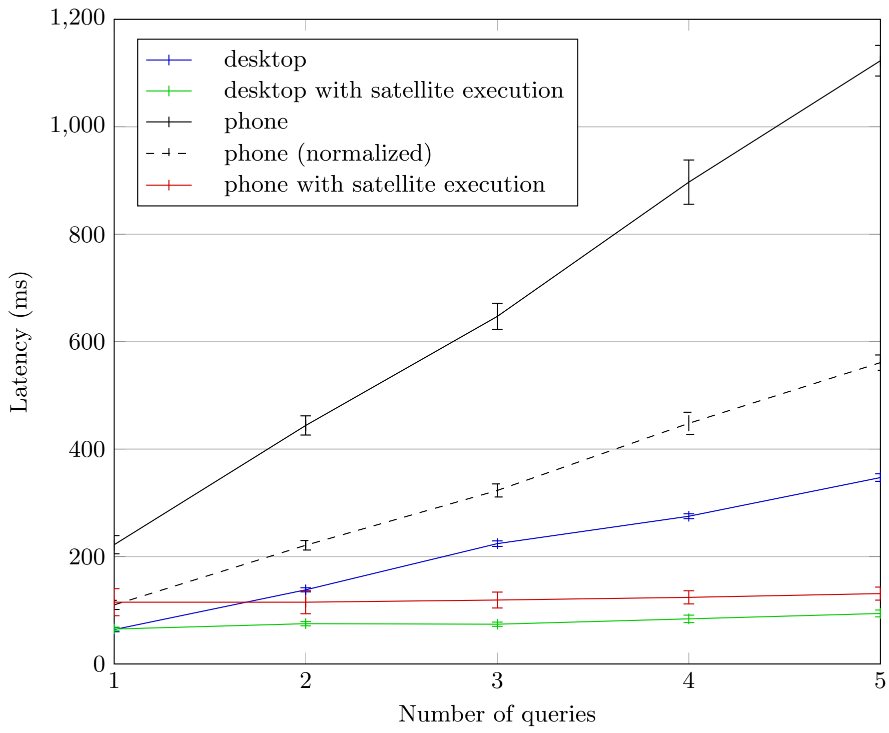
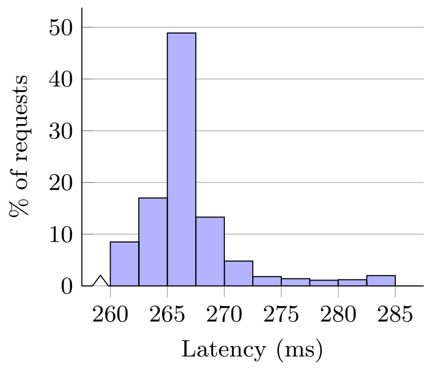


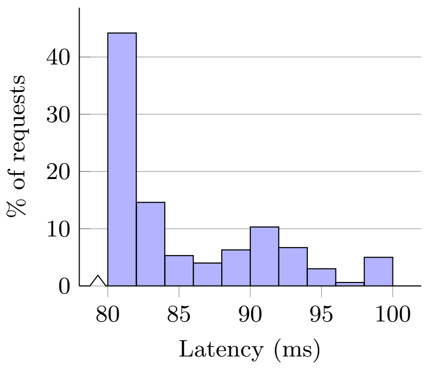
Fig. 4: Observed latency when executing a varying number of cloud database queries with and without satellite execution. Error bars show standard deviation.

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 [16]. 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 could be effective if applied in these popular applications.

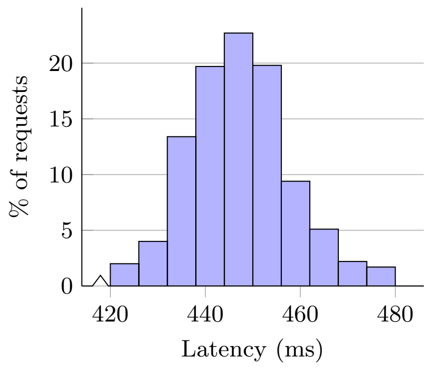
We continue with an experiment that quantifies latency when a client issues cloud database queries directly and when using satellite execution. For this we used the bag-of-queries implementation outlined in Code Listing 7 to issue queries to the cloud database. Latency when the bag contained between 1 and 5 queries is shown in Figure 4. Results are averaged over 1000 runs, for both phone and desktop, with the relay-node hosted on a t1.micro instance. 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 phone—overshadows the low cost of serializing and transferring the query bag.



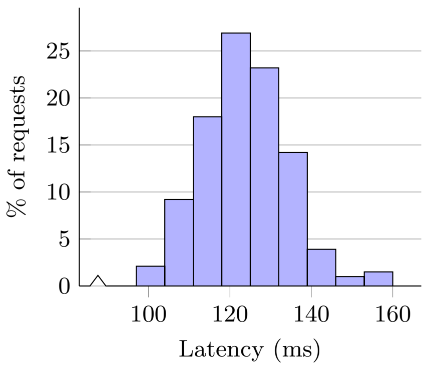
(a) Desktop: adding a friend.



(b) Desktop: adding a friend, with satellite execution.



(c) Mobile: adding a friend.



(d) Mobile: adding a friend, with satellite execution.

Fig. 5: Latencies when adding a friend to a social network, with and without satellite execution.

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 [18]. 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 4 consequently include one additional round-trip latency for phone, compared to desktop. To better convey the latency difference between phone and desktop, the figure also includes results where one round-trip latency has been subtracted from phone. Even after this normalization, phone has significantly higher latency than desktop, demonstrating the relative importance of our satellite execution technique for the mobile platform.

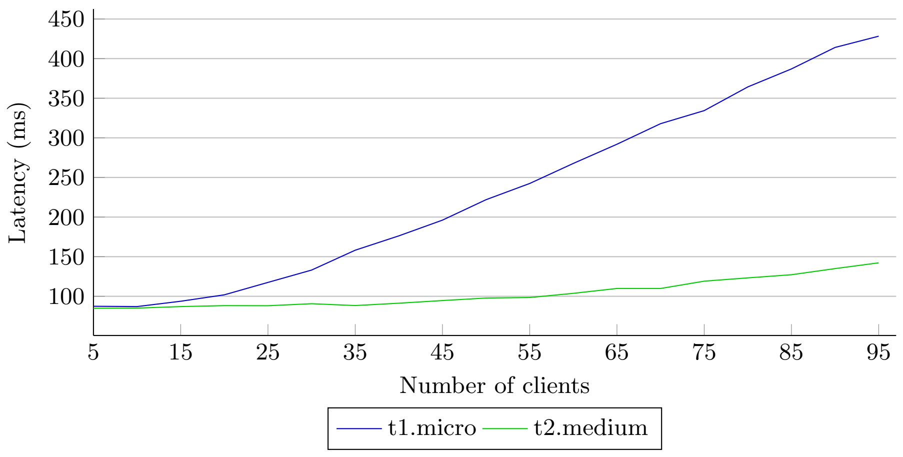


Fig. 6: Latency per bag-of-queries when increasing the number of clients that concurrently submit mobile functions to a relay-node.

The data on popular applications in Table 1 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 [12]. The addition of a friend in this network involves friend and news feed updates for both concerned parties, for a total of 4 queries. Equivalent queries were placed in our bag-of-queries and we ran the friend-add action 1000 times on both the desktop and the mobile platform, with and without satellite execution. Figure 5 illustrates latency savings. Savings due to satellite execution are pronounced; on desktop latency drops from around 265 ms to approximately 100 ms, while it drops on mobile from around 450 ms to approximately 125 ms.

On a mobile device such as a smartphone, a person uses around 24 different applications every month [15]. 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 application module service [17], the relay-node would likely be shared among many mobile devices and its capacity would be a potential issue. We therefore last consider an experiment where the relay-node serves an increasing number of clients.

In the experiment, we configured each client 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 clients, ensuring high contention for relay-node resources, in an attempt to reveal the capacity for executing mobile functions. We repeated the experiment both for t1.micro and t2.medium

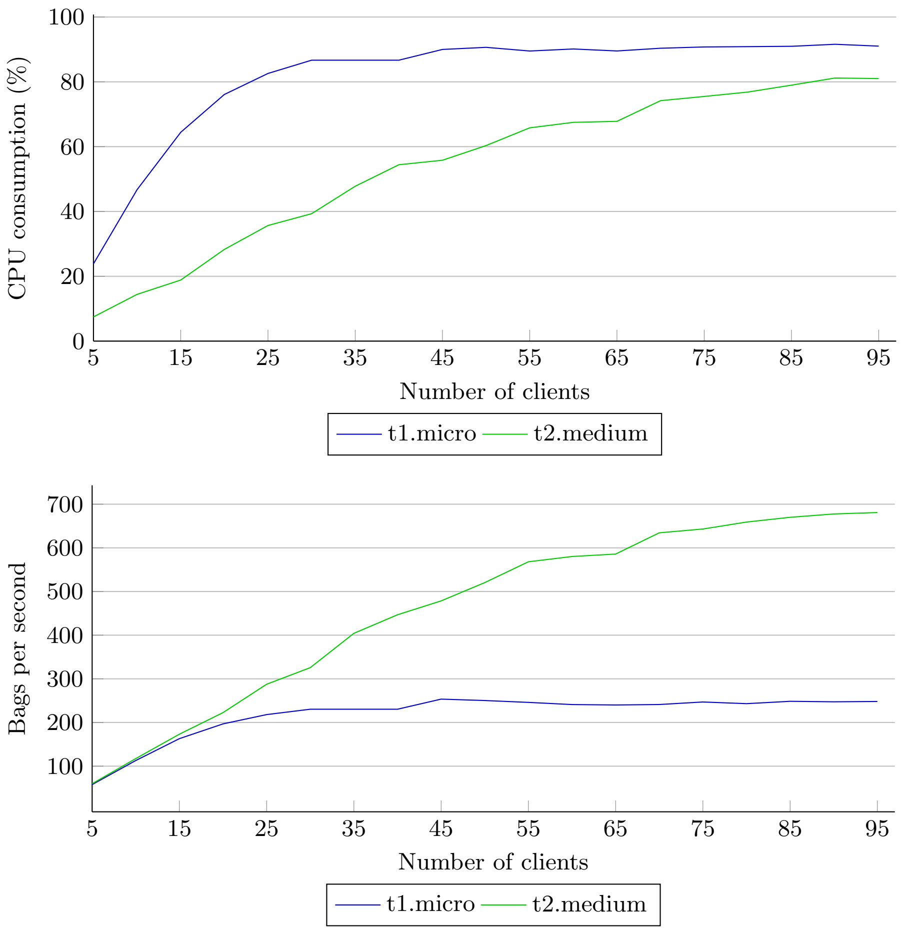


Fig. 7: CPU consumption and throughput at a relay-node when increasing the number of concurrent clients that submit mobile functions.

instances. Results are shown in Figures 6 and 7. We observe that the t1.micro instance is capable of completing around 250 bags per second before throughput levels off. As the number of clients continues to increase, each of them observes higher latency, as illustrated in Figure 6. The t2.medium instance peaks at around 700 bags per second. In Figure 7, we see a close correlation between throughput and 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 regards 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.

5 Related Work

The complexity of developing and deploying applications that span a variety of mobile devices, personal computers, and cloud services, has been recognized as a new challenge. Users expect applications and their state to follow them across devices, and to realize this functionality, one or more cloud service must usually be involved in the background. Sapphire [24] is a recent and comprehensive system that approaches this problem by making deployment more configurable and customizable, separating the deployment logic from the application logic. The aim is to allow deployment decisions to be changed, without major associated code changes. Applications are factored into collections of location-independent objects, communicating through remote procedure calls. Fabric [14] is another distributed system that aims to securely share objects among heterogeneous network nodes, and supports both data-shipping and function-shipping styles of execution.

Like these systems, Dapper 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 app store. Code is then transferred on demand from the device to the cloud, as objects move to the cloud to enjoy low-latency execution. The decision to visit the cloud or stay on the local device can be made dynamically, at run time.

With Dapper, we introduce relay-nodes in the cloud as an architectural tier between the cloud and mobile devices. Similar middle tiers have been proposed for example with Cloudlets [19], and are implemented in code-offloading systems like COMET [8], MAUI [5], and CloneCloud [4]. However, the goal of these systems is often to augment mobile devices with additional computing power, or to conserve energy [20], so the added tier may be located close to the devices, on local server machines, or wherever cheap computing power is available. In contrast, our motivation is not to offload work, but to reduce the latency of accessing cloud services, and thus the new tier sits as close to the cloud services as possible.

Concretely, Dapper reduces latency by eliminating extraneous round-trips of communication to the cloud. An alternative way to achieve that is by having cloud databases support more expressive query languages, so that more sophisticated transactions can be submitted as a single operation. Indeed, relational databases with full SQL support are part of the offerings of major cloud providers like Amazon. However, the ability to access the database via a general-purpose programming language remains appealing for its generality and flexibility. This is a lesson learned from programming models like MapReduce [6], Oivos [22], and Cogset [21], where data is accessed programmatically through user-defined visitor functions that can integrate easily with legacy code and libraries. The programming model in Dapper follows a similar philosophy, with the difference that user-defined functions are visiting a database in the cloud rather than a partition of data in a cluster.

6 Conclusion

This work focuses on the general issue of latency as a concern for applications that interact with the cloud, and looks specifically at scenarios where multiple consecutive queries are issued to a database in the cloud. Intuitively, latency can be reduced by shortening communication distances, so our idea is to move the location where queries are issued closer to the database. Since cloud databases commonly have programmatic interfaces, we implement a general mechanism for code-offloading to support this pattern.

Having a relay-node in the cloud, located in close proximity to the database service, has already proven to be a useful technique for caching, and beneficial for read-mostly database workloads [18]. Here, we extend the relay-node with functionality for *satellite execution*, allowing code that has moved temporarily from a mobile device to execute in an environment with low-latency access to cloud services. This gives benefits for additional workloads, which may include database updates.

The key characteristic that a workload must exhibit to benefit from our approach is dependencies between requests. For example, if the results from one database query are used to shape the next query, there is a dependency between the two. If there is no need for user interaction in-between requests, a whole sequence of dependent requests can be offloaded to the cloud. By eliminating extraneous round-trips of communication, this improves response times.

To estimate the potential for improvement in real applications, our evaluation examines the communication patterns of some popular applications through a black-box technique. This has yielded some indications that dependent requests 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 [12], we found specific examples. For example, a friend request results in 4 dependent database queries; when offloaded to the cloud from a phone, the completion time of a friend request dropped from 450 ms to approximately 125 ms.

Our implementation handles the practicalities of transferring assemblies of .NET code, serializing and deserializing objects, and sandboxing code that executes on the relay-node. Our evaluation gives some data points on performance: a single Amazon t1.micro instance can serve hundreds of queries per second. One such instance can thus easily handle load imposed by a large number of applications. So, we can dramatically reduce latency without disrupting application architectures and with minimal requirements for resources in the cloud.

Acknowledgements This work is part of the Information Access Disruptions (iAD) project, supported in part by the Research Council of Norway through the National Center for Research-based Innovation program. We thank our iAD colleagues for valuable feedback.

References

1. Bind DLZ, http://bind-dlz.sourceforge.net/
2. ISC Bind, https://www.isc.org/downloads/bind/
3. Application Domains: (2015), http://msdn.microsoft.com/en-us/library/cxk374d9%28v=vs.90%29.aspx
4. Chun, B.G., Ihm, S., Maniatis, P., Naik, M., Patti, A.: Clonecloud: elastic execution between mobile device and cloud. In: Proceedings of the 6th conference on Computer systems. pp. 301–314. EuroSys '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1966445.1966473
5. Cuervo, E., Balasubramanian, A., Cho, D.k., Wolman, A., Saroiu, S., Chandra, R., Bahl, P.: Maui: making smartphones last longer with code offload. In: Proceedings of the 8th international conference on Mobile systems, applications, and services. pp. 49–62. MobiSys '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1814433.1814441
6. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: Proceedings of the 6th symposium on Operating Systems Design and Implementation. pp. 137–150. OSDI '04, USENIX Association (2004)
7. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. SIGOPS Oper. Syst. Rev. 41, 205–220 (October 2007), http://doi.acm.org/10.1145/1323293.1294281
8. Gordon, M.S., Jamshidi, D.A., Mahlke, S., Mao, Z.M., Chen, X.: Comet: code offload by migrating execution transparently. In: Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation. pp. 93–106. OSDI'12, USENIX Association, Berkeley, CA, USA (2012), http://dl.acm.org/citation.cfm?id=2387880.2387890
9. Johansen, D.: 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)
10. Johansen, D., Lauvset, K.J., van Renesse, R., Schneider, F.B., Sudmann, N.P., Jacobsen, K.: A TACOMA retrospective. Software - Practice and Experience 32, 605–619 (2001)
11. Johansen, D., Marzullo, K., Lauvset, K.J.: An approach towards an agent computing environment. In: ICDCS'99 Workshop on Middleware (1999)
12. Levandoski, J.J., Lomet, D.B., Mokbel, M.F., Zhao, K.: Deuteronomy: Transaction support for cloud data. In: CIDR. pp. 123–133. www.cidrdb.org (2011)
13. Li, A., Yang, X., Kandula, S., Zhang, M.: CloudCmp: comparing public cloud providers. In: ACM SIGCOMM. pp. 1–14 (2010)
14. Liu, J., George, M.D., Vikram, K., Qi, X., Waye, L., Myers, A.C.: Fabric: A platform for secure distributed computation and storage. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. pp. 321–334. SOSP '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1629575.1629606
15. Nielsen: (2014), http://www.nielsen.com/us/en/insights/news/2014/smartphones-so-many-apps–so-much-time.html
16. Nygren, E., Sitaraman, R.K., Sun, J.: The Akamai network: A platform for high-performance internet applications. SIGOPS Oper. Syst. Rev. 44(3), 2–19 (Aug 2010), http://doi.acm.org/10.1145/1842733.1842736
17. Parse: (2015), http://www.parse.com

18. Pettersen, R., Valvåg, S.V., Kvalnes, A., Johansen, D.: 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)
19. Satyanarayanan, M.: 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)
20. Tilevich, E., Kwon, Y.W.: Cloud-based execution to improve mobile application energy efficiency. Computer 47(1), 75–77 (Jan 2014), http://dx.doi.org/10.1109/MC.2014.6
21. Valvåg, S.V., Johansen, D., Kvalnes, A.: Cogset: A high performance MapReduce engine. Concurrency and Computation: Practice and Experience 25(1), 2–23 (2013), http://dx.doi.org/10.1002/cpe.2827
22. Valvåg, S.V., Johansen, D.: Oivos: Simple and efficient distributed data processing. In: Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications. pp. 113–122. HPCC '08, IEEE Computer Society (2008), http://dx.doi.org/10.1109/HPCC.2008.105
23. Valvåg, S.V., Johansen, D., Kvalnes, A.: 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. pp. 43–50. HotTopiCS '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2462307.2462317
24. Zhang, I., Szekeres, A., Aken, D.V., Ackerman, I., Gribble, S.D., Krishnamurthy, A., Levy, H.M.: Customizable and extensible deployment for mobile/cloud applications. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). pp. 97–112. USENIX Association, Broomfield, CO (Oct 2014), https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zhang