Dynamically Loading Mobile/Cloud Assemblies

Robert Pettersen, Håvard D. Johansen, Steffen Viken Valvåg, and Dag Johansen

University of Tromsø—The Arctic University of Norway
{robert, haavardj, steffenv, dag}@cs.uit.no

Abstract. Modern distributed systems, such as mobile applications connecting to cloud-based backend services, benefits from a flexible mechanism for loading code dynamically at runtime. This paper describes LADY: a run-time system extending Microsoft's .NET platform with a reliable, low-latency, and secure mechanism for resolving and loading assemblies. We show that LADY achieves low-latency operations and high availability through its novel integration with DNS.

Keywords: Mobile, Cloud, Latency, Extensible Distributed Systems

1 Introduction

Distributed applications require their executable code to be available in local memory of each participating CPU. Traditionally, application code has been distributed and installed in advance with the underlying Operating System (OS), and fully loaded into local memory when a member process starts executing. Code can be distributed and installed manually by the machine administrators, or on a shared file system trusted by all member processes. More recently, code is commonly distributed over the Internet using some OS specific application distribution service like Ubuntu's apt-get system [8], the Google Play Store, the Apple iTunes Store, or the Windows Store.

Unlike application distribution services that require processes or even the entire OS to be restarted, run-time code injection enables long-lived distributed systems to be updated on-the-fly, minimizing service interruption. This allows systems to evolve over time as requirements change and bugs are fixed. Run-time code injection is also a foundation for extensibility in component systems like Sapphire [23] and Kevoree [3], and enable popular end-user applications like Firefox and Eclipse to be extended with third-party plugins on demand. In big-data systems like MapReduce [4], Pig [16], DryadLINQ [22], and Cogset [21], the underlying code for user-defined functions, which play a central role in distributed data processing, are typically distributed and loaded dynamically at run-time. Transferring code between processes is also a recurring requirement for actor-based distributed computing and mobile agent computing [10,11]. Whenever an object is serialized and transferred over the wire, the code required for deserialization must be somehow available at the recipient. If object schemas are allowed to evolve over time, so must the serialization code.

Dynamic run-time code injection in distributed applications demands a robust system for distributing code, managing dependencies, and resolving version conflicts. If

security related updates are to be distributed, the mechanism must also be resilient to attacks [13].

This paper presents LADY, a .NET library for loading assemblies dynamically and an associated cloud service for maintaining and looking up meta-information about assemblies.[1] LADY provides a robust and generic infrastructure for code distribution, allowing applications to locate, obtain, and dynamically load code—in the form of .NET assemblies—from remote repositories. LADY aims to do just a few things, but do them well:

- Provide a reliable and highly available service for finding up-to-date information about assemblies, including available versions and ways of obtaining them.
- Implement the required mechanisms for obtaining assemblies, for example through direct downloads or through a package installation system.
- Discover dependencies between assemblies, resolve the versioning conflicts that may result, and hide latency by caching and prefetching assembly data.
- Arrange for safe execution of partially trusted code, while retaining the ability to load new assemblies into sandboxed environments.

LADY offers a simple and unobtrusive interface focused on the central task of loading assemblies. LADY does not impose any particular architecture on the application, and can be combined and coexist comfortably with dependency injection frameworks like Microsoft's Managed Extensions Framework or Ninject, for systems that focus on extensibility. It can also be used as an auto-updater to check for bugfixed versions of libraries, a deserializer that automatically resolves references to missing assemblies, or as a utility to load and safely execute user-defined functions.

2 System Overview

LADY targets the .NET platform, and therefore revolves around assemblies: containers for compiled code, and the unit of deployment in .NET. A .NET application is compiled into one or more assemblies. Each application has exactly one *main assembly*, which contains that application's entry point, and is stored as an executable .EXE file. Non executable assemblies are stored as .DLL files.[2]

Assemblies can include functionality from other assemblies by referencing them. Assemblies can be cryptographically signed by their creators, which lets the .NET runtime verify that they are authentic before loading them. When an assembly is signed, the public key of the signer is combined with the assembly's short name, its localization culture, and its version number to produce a so-called *strong name*. Strong names are globally unique and therefore allow assemblies to reference each other by name without ambiguity.

[1] LADY is an acronym for Loading Assemblies Dynamically.

[2] Both .EXE and .DLL files have the same Portable Executable file format.

2.1 The Assembly Lookup Service

A central feature of LADY is its globally available and resilient Assembly Lookup Service (ALS), which resolves assembly names to URIs. LADY can also determine if an assembly has been superseded by a more recent version, and provides the functionality for obtaining assemblies in a number of ways, for example by downloading them via HTTP. Additional features include caching of assemblies, automatic resolution of assembly references (for example during deserialization), prefetching of assemblies based on dependencies, and creation of sandboxed environments to execute partially trusted code.

LADY stores meta-information about assemblies in a cloud database. To load an assembly and proceed with execution, an application may have to wait for a database lookup to complete. We therefore value predictable and low-latency lookup performance. Currently, we use Amazon's DynamoDB [5] as our database backend as it has an official API for C#, boasts scalability, and offers predictable performance. LADY does not rely on other advanced database features and can therefore easily be adapted for other database systems.

LADY manages a database D containing information on all known assemblies. Each assembly m has a name, a culture, a public-key token, and a version number. The *Strong Name (SN)* of m uniquely identifies m in the universe Ψ of all assemblies, and is defined by the tuple:

$$\mathrm{SN}(m) = (m.name, m.culture, m.publicKeyToken, m.version)$$

We define the *Packet Name (PN)* of m similarly to the strong name tuple, but without the version number:

$$\mathrm{PN}(m) = (m.name, m.culture, m.publicKeyToken)$$

Any two assemblies $m,n \in \Psi, \mathrm{SN}(m) \neq \mathrm{SN}(n)$ are members of an implicitly related sequence of assemblies if they have the same packet name $\mathrm{PN}(m) = \mathrm{PN}(n)$. Because version numbers are assumed to be monotonically increasing, packet names define ordered set of assemblies that relate in name, culture, and public key tokens; typically used for different version of the same code-base as it evolves over time.

Let $D \subseteq \Psi$ be the set of all assemblies known to LADY. For each assembly $m \in D$, LADY maintains two types of information: base records and assembly records. *Base records* maps the packet name $\mathrm{PN}(m)$ to the strong names of all assemblies $m' \in D$ with $\mathrm{PN}(m) = \mathrm{PN}(m')$. This enables LADY to support wildcard queries for specific version on an assembly, like for the most recent one in D. *Assembly records* maps $\mathrm{SN}(m)$ to the network location and protocol for downloading the assembly code for m. For instance, the record may contain a download URL, or a NuGet[3] package identifier and version.

2.2 Security

The *public-key token* of assembly m is defined by the .NET framework as an 8-byte hash of the public key matching the private key used to sign m; commonly displayed as

[3] NuGet is a package management system, closely integrated with Microsoft Visual Studio.

a 16-digit hexadecimal number. Since the public-key token is determined by the signer of m, and also incorporated into $\mathrm{SN}(m)$, it is not possible to modify m without knowing the private key (or breaking cryptography building blocks). With access to the source code for m, the code can be recompiled and signed it with a different key. However, the resulting assembly m' would have a different packet name $\mathrm{PN}(m') \neq \mathrm{PN}(m)$, and thus also a different strong name. As such, an attacker cannot successfully trick correct processes to execute m' instead of m.

While the assembly naming scheme protects against malicious tampering with the code, we would also like to guarantee that LADY can provide genuine and valid download locations. Even if an attacker cannot manufacture fake assemblies, he could potentially register an assembly with invalid meta-information, rendering it unobtainable through LADY. To guard against such attacks, we require registrations of assemblies to be in the form of signed messages, where the message signer's public key must correspond to the public key token of the assembly in question. This ensures that the person or program registering the assembly is the same as the signer of the assembly, and third parties cannot register invalid information about assemblies.

2.3 Assembly Registration

To add new assemblies to LADY, we provide the `register` command-line utility. The `register` utility does not directly update LADY's cloud database D, but instead constructs a registration message $\{REG,\ m\}_k$, signed with key k, provided by the sofware vendor. The signed message can then be sent to LADY. Upon receiving $\{REG,\ m\}_k$, LADY verifies that k matches $m.publicKeyToken$, before adding m to D. For instance, to register the `MyLib` assembly, the vendor will run:

```
$ lady register -a MyLib.dll -p MyLib -v 1.2 -k mykey.pfx
```

Here, the assembly file is specified with the `-a` option. The utility uses reflection to extract the strong name $\mathrm{SN}(\texttt{MyLib.dll})$. The `-p` and `-v` options specify a NuGet package identifier and version, respectively, so the assembly will be registered as obtainable by using NuGet to install version 1.2 of the MyLibrary package. Finally, the `mykey.pfx` file, specified with the `-k` option, contains the key pair for signing the registration message. If the public key does not match the public key token of the assembly, registration will fail. The `register` utility is implemented to be suitable for scripting and integration into existing build systems.

We have settled initially on this model where assemblies must be explicitly added to LADY by their vendors. It would be possible to create automated tools for registering assemblies that have been created by others. For example, we could integrate with existing package management systems like NuGet and scan all newly uploaded packages for strong-named assemblies, automatically registering them with LADY. This could improve the coverage of our lookup service, and possibly be more convenient for developers, but we have deferred that investigation to future work.

2.4 Loading Assemblies Explicitly

LADY provides the `LoadAssembly` method to applications for explicit loading of assemblies at runtime. For example usage, consider the configuration parser in Code Listing 1.

Code Listing 1: Example to illustrate dynamic loading of an assembly, and how to access its functionality.

```
using YamlDotNet.Serialization;
using YamlDotNet.Serialization.NamingConventions;

class ConfigParser
{
   static readonly ILady lady = LadyFactory.Init();
   static readonly Assembly yaml = lady.LoadAssembly(
      name: "YamlDotNet", publicKeyToken: "ec19458f3c15af5e", version: "3.*");

   public class Config
   {
      public string ConferenceName { get; set; }
      public DateTime Deadline { get; set; }
   }

   public static Config AccessUsingReflection(string data)
   {
      var namingConvention = yaml.NewInstance(
         "YamlDotNet.Serialization.NamingConventions.CamelCaseNamingConvention");
      dynamic d = yaml.NewInstance("YamlDotNet.Serialization.Deserializer",
                                   null, namingConvention, false);
      return d.Deserialize<Config>(new StringReader(data));
   }

   public static Config StaticallyTypedAccess(string data)
   {
      var d = new Deserializer(namingConvention: new CamelCaseNamingConvention());
      return d.Deserialize<Config>(new StringReader(data));
   }
}
```

Here, the code calls `LoadAssembly` at an early point in the program execution[4] to load the YamlDotNet library, identified by its assembly name and public key token. The culture is left unspecified and defaults to "neutral". The latest release with major version 3 is requested by specifying "3.*" as the version number.

YamlDotNet provides functionality for parsing of YAML—a human-friendly serialization language commonly used in configuration files. Bugs in configuration parsing can be unpleasant and can potentially render the application exploitable. By loading the code dynamically with LADY, the application can ensure that it always has the latest available version of YamlDotNet library, thereby picking up any bugfix releases promptly and automatically. It will not be necessary to deploy a new version of the ap-

[4] In this example, the call happens in the static constructor of the `ConfigParser` class.

plication just because a bug has been discovered and fixed in one of the libraries that it depends on.

Once an assembly is loaded, its functionality can be accessed programmatically in two ways. The first approach is to use reflection to instantiate objects and invoke methods. This is exemplified in the method `AccessUsingReflection`, which parses a YAML string into a `Config` object. The implementation instantiates a `Serializer` object using reflection, before invoking its `Deserialize` method. This approach certainly works, but there are some factors that make it cumbersome:

1. Types must be specified as strings with fully-qualified type names: a verbose and error-prone task. The verbosity stacks up when multiple types are involved; in the example, the `Serializer` constructor requires a `CamelCaseNamingConvention` object, which must be instantiated first.
2. Constructor arguments are specified as `object` instances, without static type checking. Method calls have similar constraints. The invocation of `Deserialize` looks superficially as if it might be type-checked by the compiler, but in fact the code relies on `dynamic` variables, which are assumed to support any and all operations, and defer actual type checking until run-time.
3. Named and default arguments cannot be used. Combined with the lack of static type checking, this often leads to long lists of `null` arguments where any non-default arguments must be positioned with great care.
4. Finally, reflective invocations add overhead, which may be an issue if they end up sitting on the critical path.

Also note that the `NewInstance` method used in this example is itself an extension method that we have implemented as a convenience in a utility library. `NewInstance` fills in default values for various optional hooks and packs the constructor arguments into an array. Without relying on such helpers, the object instantiation code would have to be even more verbose.

The drawbacks of reflection might call into question the practical utility of loading assemblies dynamically. Fortunately, there is a way to get the best of both worlds, and benefits from static type checking and related IDE features like code completion while still using LADY under the hood. This second approach is to compile the application with the most recent assembly versions that are available at build time, and override the assembly resolution mechanism at run-time so that LADY gets a chance to load any newer versions that may have been released since then.

The `StaticallyTypedAccess` in Code Listing 1 demonstrates this approach. The method is similar to `AccessUsingReflection`, but with the clarity and safety of normal syntax, with type checking, and without the overhead of reflective calls. This works because the compiler has access to the YamlDotNet assembly at compile time, but also means that a reference to that specific version of YamlDotNet is included in the application's assembly. However, assembly references are not resolved immediately when an application starts. The .NET runtime resolves assemblies on demand, when a method that references the assembly is first entered. On startup, LADY hooks into the assembly loading mechanism by overriding certain event handlers, and therefore gets to decide how exactly to resolve an assembly reference. By the time `StaticallyTypedAccess` is

invoked, LADY has already been instructed to load the *latest* version of the YamlDotNet assembly, so that is the version that will be used.

2.5 Loading Referenced Assemblies

In additional to explicitly loaded assemblies, as described in Section 2.4, LADY supports loading assemblies by references in the code. For example, an application might load a plug-in assembly through LADY, and the plug-in might contain references to other assemblies that have not been loaded, or even installed. Another scenario that may trigger assembly resolution is during object deserialization when data contain references to types defined in unresolved assemblies. A prime advantage of LADY is that any blob of serialized data can be deserialized at any node and at any time, so long as all of the referenced assemblies have been registered with LADY.

Resolving assemblies on demand raises the question of how to deal with conflicting versions. If plug-ins A and B both reference assembly C, but demand different versions of C, or if two blobs of data were serialized with different versions of an assembly, a potential conflict will result. One technical possibility is to load multiple versions of the same assembly. However, this is not a recommended practice, due to the confusion that may arise when types have the same name but different identities [15].

In some cases, the right thing to do is simply to load the most recent assembly version that exists. Of course, this only works for versions that are backwards-compatible. There is a standard called *semantic versioning* [1] that would resolve this issue if it was adopted universally. With semantic versioning, the major version number is bumped whenever a backwards-incompatible change is introduced. However, a 2014 survey indicates that this standard remains to be widely adopted [19]. Therefore, LADY takes a more conservative approach and does not attempt to infer automatically if two versions of an assembly are compatible. Instead, we rely on hints from the client, in the form of a compatibility policy, which is a simple boolean-valued function that may be specified programmatically. Whenever LADY must determine if a given pair of assembly versions should be considered compatible, it consults the compatibility policy by invoking this function. The default compatibility policy is a slightly stricter version of semantic versioning: if both the major and the minor version numbers are equal, the assemblies are considered compatible. (Build and revision numbers may still differ.)

Armed with this concept of compatibility policies, LADY takes the following approach to assembly resolution: the assembly lookup service is first queried to retrieve all known versions of the assembly in question, and the most recent version that is compatible with the requested version is then selected. LADY then proceeds to obtain and load this specific version of the assembly. For example, if an assembly is registered with versions 1.0.1, 1.0.2, and 1.0.3, and three plug-ins each reference one of these versions, then the actual version that will be loaded (under the default compatibility policy) is 1.0.3, regardless of the order in which the plug-ins are loaded.

2.6 Loading Partially Trusted Code

While plug-ins might be considered trusted code by some applications, there are many cases where applications wish to load and execute partially trusted code with a limited

Code Listing 2: Example code using LADY to load partially trusted user-defined functions inside a sandbox.

```
using Microsoft.Hadoop.MapReduce;

class MapReduceSandbox : LadySandbox
{
    public MapReduceSandbox(object x) : base(x) { }

    public override void Play()
    {
        // Load an assembly with partially trusted UDFs
        Assembly myUDFs = lady.LoadAssembly(name: "MyUDFs",
            publicKeyToken: "8c11fe16618d1673", version: "*");
        var mapper = myUDFs.NewInstance("WordCountMapper") as MapperBase;
        var reducer = myUDFs.NewInstance("WordCountReducer") as ReducerCombinerBase;
        // The mapper and reducer may now be invoked in relative safety; they
        // cannot access the file system, network, environment, etc.
    }
}

class MapReduceProgram
{
    static void Main(string[] args)
    {
        LadyFactory.Init().MakeSandbox(typeof(MapReduceSandbox)).Play();
    }
}
```

set of permissions. For example, distributed data processing models like MapReduce rely on user-defined functions for flexibility and expressiveness. When invoking these functions, it is prudent to do so from a sandboxed environment with restricted capabilities for hazardous actions like network and file I/O. On the surface, this appears to preclude the use of LADY from user-defined functions, since network and file I/O are needed to locate and obtain an assembly, and a full, unrestricted permission set is required in order to override the assembly resolution mechanism.

LADY resolves this problem by offering a *sandbox* abstraction based on .NET application domains [14]. Application domains provide an isolation boundary for security, reliability and versioning, and for loading assemblies. They 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 additional application domains to further separate and isolate execution of code.

LADY must be initialized (using the `LadyFactory` class) exactly once per process, and from a fully trusted application domain—typically the initial application domain that is created on startup. This singular instance of LADY thus executes with unrestricted permissions, as required. However, users may create additional sandboxes using the

`MakeSandbox` method, as exemplified in Code Listing 2, where the user-defined functions required for a MapReduce job are loaded inside a sandbox. The sandbox is a partially trusted application domain that is initialized with a figurative umbilical cord that leads back to the fully trusted application domain. Concretely, the application must implement a subclass of `LadySandbox` with a single-argument constructor that passes on a special proxy object to its base class. The proxy object is named `x` in the example, and constitutes the umbilical cord.

Inside a sandbox, execution starts in the `Play` method. Any `LoadAssembly` calls made inside the sandbox get routed back to LADY using cross-domain remote method calls on the proxy object. LADY will determine which version to load, as described in the previous section, and retrieve the assembly data, either directly from its cache, or by first obtaining the assembly. The assembly data is then passed back to the sandbox, where it is loaded into the partially trusted application domain. Assemblies that must be loaded due to code references or during deserialization are handled similarly. This approach effectively grants sandboxes full capabilities with regards to loading of assemblies, so long as this happens through LADY. The implementation details of how to communicate across application domains are hidden. All sandboxes also share the benefit of a common cache.

3 DNS Integration

With the assembly lookup service, LADY adds a level of indirection to loading assemblies. This relieves applications of various responsibilities, and enables several useful applications. It also raises some important concerns:

Availability. If the assembly lookup service becomes unavailable, applications may also experience various forms of unavailability. For example, it might not be possible to start a scheduled MapReduce job because the assembly that contains the required user-defined map function cannot be located. (LADY does maintain a client-side cache, but it could also be missing there.)

Scalability. LADY is designed to serve numerous application instances running in many different locations all over the globe. The aggregated number of queries for assembly information is expected to be large. While cloud databases like DynamoDB generally provide great scalability, this does come with a monetary cost. As the volume of requests grows, the financial cost of operating the assembly lookup service could become prohibitive.

Latency. Applications may have a tendency to load assemblies sequentially, either as a result of logical dependencies or due to the sequential nature of their execution. Any extra latency incurred when an assembly is loaded may thus stack up and result in unwanted user-perceptible delays, for example on application startup. We should therefore strive to minimize the latency of individual assembly lookups.

To address these three concerns, we have integrated the assembly lookup service with the Domain Name System (DNS). Given that registration of new assemblies is

expected to be a relatively rare event, the assembly lookup service's workload is almost read-only. This implies that caching can be an effective way to reduce both load and latency, and DNS is a globally distributed cache readily available and with extremely high availability. While the most common use of DNS is to associate globally unique host names with IP addresses, we use it to associate assemblies strong name with their meta-data, such that $\forall m \in D, \text{Resolve}(\text{SN}(m)) \implies m$. This approach addresses all three concerns above, since DNS is globally available, will significantly alleviate the load on the cloud database, and can generally be accessed with low latency.

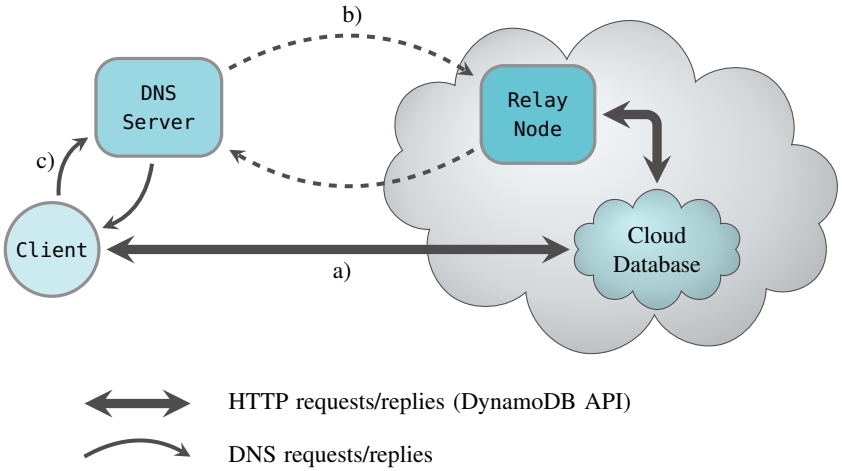

Fig. 1: Assembly lookups with and without DNS integration.

For the DNS integration we rely on our previous work with the Jovaku system [17]. Jovaku mirrors database keys as labels in the DNS namespace. Database lookups can then be translated into DNS request on the client side. A relay-node is set up close to the cloud database that performs the opposite translation. Figure 1 shows how this works in *(a)* the baseline case where we have no DNS integration, *(b)* the case where we miss the DNS cache, and *(c)* the common case where we hit the DNS cache. This approach is very effective at reducing latency for read-mostly workloads like the one exhibited in LADY [17].

4 Evaluation

Many of the reasons to adopt LADY are anecdotal as it is hard to quantify benefits like flexibility and extensibility. However, there are concrete performance benefits, as we will demonstrate in this section. As a case study we rely on our previous work on *satellite execution* [18]: a technique to reduce latency for applications that interact repeatedly with cloud services by temporarily offloading code so it executes in closer

proximity to the cloud. For this, we developed the *mobile functions* a programming abstraction, and an *execution server* capable of receiving and executing such functions.

In our original implementation of satellite execution, referred to as *baseline*, the execution server receives mobile functions as serialized objects. Before the mobile function's entry point can be invoked, the objects must be deserialized and allowed to execute. Deserialization can fail if an unresolvable assembly reference is encountered. In the baseline implementation, we handled this and other assembly resolution errors by returning an error to the client. The client would then have to upload the missing assembly to the execution server, before making a new attempt to offload the mobile function.

The baseline design for offloading mobile functions was grounded in two assumptions: (1) the client will have the code for any assemblies that are referenced by its mobile functions, and (2) the execution server has some means to resolve any assembly resolution errors it encounters. Although both assumptions are reasonable, they can cause excessive back-and-forth communication between the client and the execution server for mobile functions that depend on multiple assemblies.

Refactoring our baseline implementation to use LADY made the implementation of both the execution server and the client simpler, and the services became more robus. The modified execution server can deserialize mobile functions without having to interact with the client, trusting LADY to resolve assemblies. Similarly, a mobile function can be invoked through its entry point, and any referenced assemblies will be loaded through LADY. Additionally, we use LADY's sandboxing support, as described in Section 2.6, to isolate the execution of mobile functions in a separate application domain, minimizing the potential for disruption by misbehaving mobile functions.

Table 1: Machines involved in evaluating the effectiveness of using LADY for reducing latency in satellite execution, along with the latency and hop count to the desktop client located in Tromsø.

Location	**EC2 type**	**Ping Latency**	**# Hops**
Ireland	t2.medium	64 ms	15
California	t2.medium	155 ms	17
Singapore	t2.medium	339 ms	20
Sydney	t2.medium	365 ms	12

We envision diverse applications for LADY, that may exhibit many different access and usage patterns. Performance also depends on how an application is deployed geographically, since this affects the latency to access both the execution server and DNS. Therefore, we use synthetic workloads in our experiments, and run experiments from multiple geographical locations, comparing the baseline and LADY implementation.

We set up experiments with the execution server hosted on Amazon EC2 nodes in various geographically zones. We chose Ireland, California, Singapore and Sydney to exhibit variations in the routing distance to the client located in Norway. The EC2 nodes was equipped with 4 GB memory and a dual-core 2.5 GHz 64 bit vCPU, and the client was a desktop machine equipped with 64 GB memory and a quad-core Intel Xeon E5-1620 3.7 GHz CPU. Typical observed ping latency and hop count between the client and the execution nodes is illustrated in Table 1.

We stored a set of assemblies in a separate table in the DynamoDB database in each of the locations, acting as our package management system, and registered these assemblies with LADY. We then tried executing mobile functions with a varying number of assembly dependencies that would be resolved sequentially as the execution progressed.

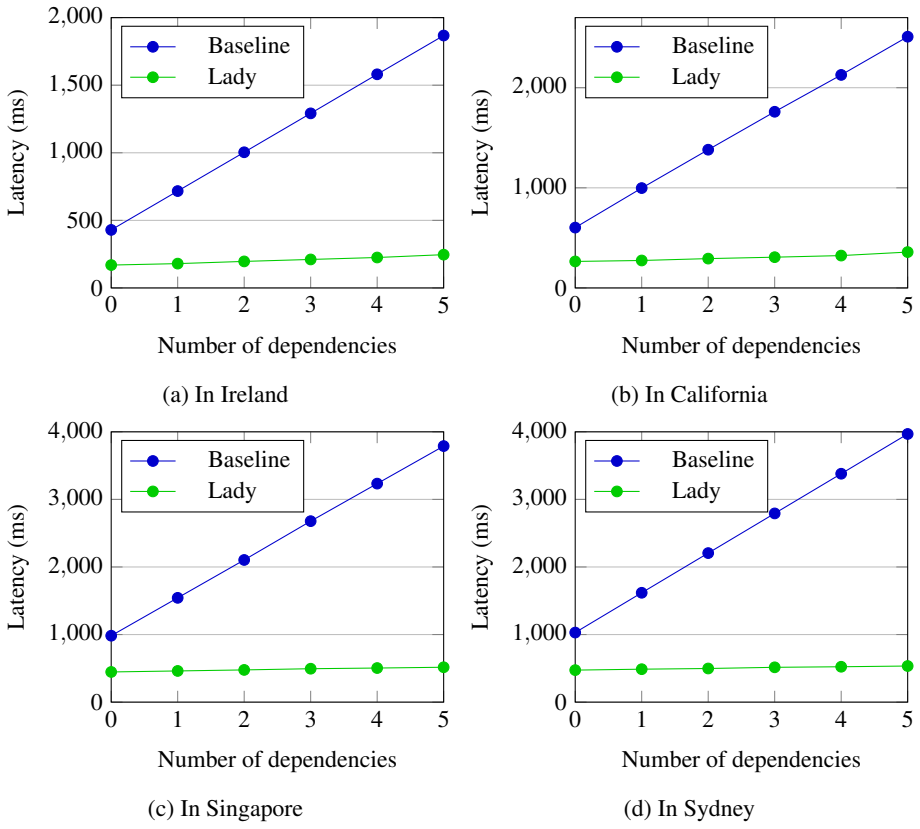

Fig. 2: Observed mean latency when executing a mobile function with a varying number of assembly dependencies with and without LADY for various geographical locations.

Figure 2 shows the difference in latency between our baseline implementation of satellite execution, and the refactored implementation using LADY. Given that the mo-

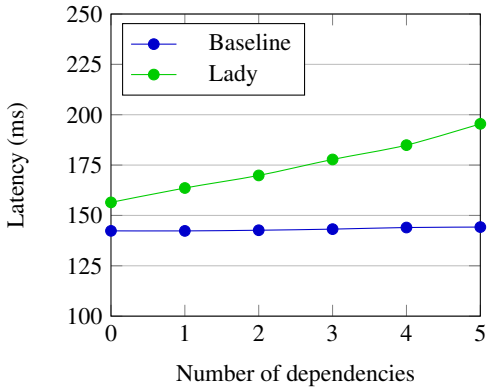

Fig. 3: Observed mean latency when executing a mobile function with all assembly data present, where LADY performs DNS lookups to ensure that the latest versions will be loaded, and the baseline does not check for updated versions.

tivation for satellite execution is to reduce latency, the observed performance benefits of using LADY are highly significant. Even when a mobile function has no additional dependencies beyond its own assembly, it saves one round-trip between the execution server and the client. By comparison, LADY is only making low-latency DNS requests to resolve assemblies, coupled with lookups to DynamoDB to retrieve the assembly data. The reason that we save latency in this scenario is two-fold:

1. We substitute long round-trips between Norway and Ireland with much faster DNS lookups.
2. Assembly data is stored in the cloud, instead of at the client. Since we need to load the assemblies at a node in the cloud, this data placement is more optimal.

Not every application that employs LADY will benefit from the same fortuitous circumstances. For example, if all assemblies are obtained in advance, LADY will add the overhead of one DNS lookup for each resolved assembly. In exchange, LADY guarantees that the most recent assembly version is found. Whether this is a reasonable trade-off depends on the application's requirements for flexibility and extensibility. Figure 3 shows the overhead added by LADY for the satellite execution scenario, when the experiment is set up so that all assembly data has been obtained in advance. The overhead is proportional to the number of dependencies, as each dependency adds another DNS lookup. The actual resource requirements for performing a DNS lookup are negligible.

It is worth noting that the overhead illustrated in Figure 3 is the worst-case scenario. The mobile function used in the experiment does not do any useful work, and will be stalled while waiting for LADY to check for updated assembly versions. In a more realistic scenario the mobile function will do useful work while LADY checks for updated assemblies in the background, and might not need to stall when the next dependency boundary is crossed.

5 Related Work

Package-management systems backed by online code repositories have become common for deploying applications in many modern systems. For instance, popular operating systems like the Linux based Ubuntu and Debian systems rely on the Advanced Package Tool (APT) for software installation, upgrade, and dependency resolving [8]. To host the code online, these communities depend on donated third-party servers, known as mirrors, to distribute their software to millions of end-users. Although, these software mirroring infrastructures lack the mechanisms to deal with the wide-range of faults that can occur, solutions for resilient software mirroring has been demonstrated [12,13]. Systems distributed commercially, like Microsoft Windows, often come equipped with proprietary mechanisms for distributing software updates [7] and are generally less vulnerable to intrusions.

In the framework for code updates described by [9], semi-automatically generated software patches include both the updated code and the code for making the transition safely. By using the Typed Assembly Language, these patches can consist of verifiable native code, which is highly beneficial to system safety.

However, these systems are primarily geared towards *installing* applications into a relatively static environment. LADY goes a step further and supports dynamic loading of code into running applications. A package management system generally aims to ensure that all prerequisites for an application—e.g., the assemblies that it may depend on—are installed before launching the application. LADY takes a different approach and obtains these assemblies on demand, if and when they are referenced and must be loaded. In some cases, the assemblies that may be required are truly unpredictable, as in our satellite execution system, and LADY can solve a problem that package management systems fail to address.

The problems of applying dynamic updates of running programs is well known and has been the subject of research for several decades [20]. DYMOS [2] is perhaps the earliest programming system that explores the ideas of dynamic updates of functions, types, and data objects. DYMOS is based on the StarMod extension of the Modula language, and it is unclear to what extent the proposed mechanisms are applicable to modern application platforms like .NET. Other programming languages, like Standard ML, have also been demonstrated to support dynamic replacement of program modules during execution [6]. Our approach specifically targets the .NET platform and leverages the capabilities of the .NET application domains and their customizable assembly resolution mechanism.

The general complexity of developing and deploying modern distributed applications, which 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 services must usually be involved in the background. Sapphire [23] 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.

We envision LADY as a particularly useful sidekick for the design and implementation of this new generation of highly flexible and extensible distributed systems. By facilitating the on-demand resolution of assemblies, system architectures can make the simplifying assumption that all participants will share a common code base, and enjoy greater freedom in their deployment decisions.

6 Conclusion

Loading code on-demand implies that *the distribution of code is decoupled from the distribution of state*, so that the code does not have to be propagated through the same communication substrate as the data. Such decoupling enables construction of adaptable application-level communication substrates, like distributed hash tables and gossip protocols, that can host multiple concurrent application, without having to standardize messaging format or requiring the sender to include deserialization code in all messages.

LADY makes .NET code available in a globally accessible online service so that it can be referenced unambiguously by name and retrieved on demand at run-time. Although strong-named .NET assemblies already have globally unique names, the current .NET platform lacks the ability to load code in many contexts. LADY fills this gap by creating a low-latency and reliable lookup service for assemblies, and by implementing the mechanisms for obtaining code on-demand during program execution. With the separation of concerns provided by LADY, the design of many distributed systems could be simplified, since code would be retrieved on demand via a mechanism independent of the data substrate. Our satellite execution refactoring in section 4 illustrates well how LADY can simplify the design of distributed systems, to improve extensibility and serve as a convenient foundation for mobile code.

References

1. Semantic Versioning, http://semver.org/
2. Cook, R.P., Lee, I.: DYMOS: A dynamic modification system. vol. 8, pp. 201–202. ACM, New York, NY, USA (Mar 1983), http://doi.acm.org/10.1145/1006140.1006188
3. Daubert, E., Fouquet, F., Barais, O., Nain, G., Sunye, G., Jezequel, J.M., Pazat, J.L., Morin, B.: A models@runtime framework for designing and managing service-based applications. In: Software Services and Systems Research - Results and Challenges (S-Cube), 2012 Workshop on European. pp. 10–11 (June 2012)
4. 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)
5. 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. In: Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles. pp. 205–220. SOSP '07, ACM (2007), http://doi.acm.org/10.1145/1294261.1294281
6. Gilmore, S., Kirli, D., Walton, C.: Dynamic ML without dynamic types. Technical report, University of Edinburgh (1997)

7. Gkantsidis, C., Karagiannis, T., Rodriguez, P., Vojnović, M.: Planet scale software updates. ACM SIGCOMM Computer Communication Review 36(4), 423–434 (2006)
8. Hertzog, R., Mas, R.: The Debian Administrator's Handbook. Freexian SARL, https://debian-handbook.info/, first edn. (2006)
9. Hicks, M., Moore, J.T., Nettles, S.: Dynamic software updating. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. pp. 13–23. PLDI '01, ACM, New York, NY, USA (2001), http://doi.acm.org/10.1145/378795.378798
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. Johansen, H., Johansen, D.: Resilient software mirroring with untrusted third parties. In: Proceedings of the 1st ACM workshop on hot topics in software upgrades (HotSWUp) (Oct 2008)
13. Johansen, H., Johansen, D., van Renesse, R.: 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)
14. Microsoft: Application Domains (2015), http://msdn.microsoft.com/en-us/library/cxk374d9%28v=vs.90%29.aspx
15. Microsoft Developer Network: Best Practices for Assembly Loading. Microsoft, .NET Framework 4.6 and 4.5 edn. (2016), https://msdn.microsoft.com/en-us/library/dd153782(v=vs.110).aspx
16. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data. pp. 1099–1110. SIGMOD '08, ACM (2008), http://doi.acm.org/10.1145/1376616.1376726
17. 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)
18. Pettersen, R., Valvåg, S.V., Kvalnes, A., Johansen, D.: 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)
19. Raemaekers, S., van Deursen, A., Visser, J.: Semantic versioning versus breaking changes: A study of the maven repository. In: Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on. pp. 215–224 (Sept 2014)
20. Segal, M., Frieder, O.: On-the-fly program modification: systems for dynamic updating. Software, IEEE 10(2), 53–65 (March 1993)
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. Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Gunda, P.K., Currey, J.: DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In: Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation. pp. 1–14. OSDI'08, USENIX Association (2008), http://dl.acm.org/citation.cfm?id=1855741.1855742
23. 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 (10 2014), https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zhang