LADY: Dynamic Resolution of Assemblies for Extensible and Distributed .NET Applications

Steffen Viken Valvåg, Robert Pettersen, Håvard D. Johansen and Dag Johansen
University of Tromsø — The Arctic University of Norway, Tromsø, Norway

Keywords: Mobile, Cloud, Latency, Extensible Distributed Systems.

Abstract: Distributed applications that span mobile devices, computing clusters, and the cloud, require robust and flexible mechanisms for dynamically loading code. This paper describes LADY, a system that augments the .NET platform with a highly reliable mechanism for resolving and loading assemblies and arranges for safe execution of partially trusted code. Key benefits of LADY are the low latency and high availability achieved through its novel integration with DNS.

1 INTRODUCTION

Distributed applications require their code to be made available locally in the memory of enlisted processors. Traditionally, the code to be executed is installed in advance in the operating system that hosts the member processes, either manually or using some application-distribution mechanism like Ubuntu's apt-get system (Hertzog and Mas, 2006). In a computing cluster, code could be installed in a shared file system and readable by all nodes. In the case of mobile devices, the operating system is commonly tied to some vendor-specific application store, like the Google Play Store, Apple's iTunes Store, or the Windows Store.

However, for long-lived distributed systems, perhaps spanning both mobile device, local computing clusters, and cloud services, run-time code injection may be the only feasible approach to evolve the system as requirement change and bugs must be patched. This demands a robust system for distributing code, managing dependencies, and resolving version conflicts, and there must be dynamic mechanisms for obtaining and load code from remote sources. The ability to dynamically obtain and load code is useful in a wide range of contexts:

- As a foundation for extensibility, for example in component systems like Sapphire (Zhang et al., 2014) and Kevoree (Daubert et al., 2012), or in plug-in systems embedded in extensible applications like IDEs and web browsers, where third party code must be installed.
- In systems like MapReduce (Dean and Ghemawat, 2004), Pig (Olston et al., 2008), DryadLINQ (Yu et al., 2008), and Cogset (Valvåg et al., 2013), where user-defined functions play a central role in distributed data processing, and the code that implements those functions must be distributed to a number of nodes.
- To enable mobility, for example with actor-based distributed computing, mobile agents (Johansen et al., 2001), or code offloading. Transferring code between nodes is a recurring requirement in these scenarios.
- As a way to improve security, by facilitating the deployment of security fixes, which must be rapid to reduce the window of vulnerability (Johansen et al., 2007).
- Whenever serialization is used to transfer or store objects. Serialized objects can be passed around in any number of complicated ways, or stored in files or databases to be read at a later time by another process. To successfully deserialize an object, its code must also somehow be available.

Over the years, we have built many research systems where the requirement to dynamically distribute code was present, in one or more of the contexts outlined above. With the work presented here, we have focused exclusively on solving this one recurring requirement in a comprehensive and reusable way.

This paper describes LADY[1], a .NET library for loading assemblies dynamically and an associated

[1] LADY is an acronym for Loading Assemblies Dynamically

Valvåg, S., Pettersen, R., Johansen, H. and Johansen, D.
LADY: Dynamic Resolution of Assemblies for Extensible and Distributed .NET Applications.
In *Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016) - Volume 2*, pages 118-128
ISBN: 978-989-758-182-3
Copyright © 2016 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

cloud service for maintaining and looking up meta-information about assemblies. 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 that focuses on the central task of loading assemblies, and does not impose any particular architecture on the application. For example, it 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 be used as an auto-updater to check for bugfixed versions of libraries, as a deserializer that automatically resolves references to missing assemblies, or as a utility to load and safely execute user-defined functions.

2 LADY OVERVIEW

LADY targets the .NET platform, and therefore revolves around assemblies, which are containers for compiled code and the fundamental unit of deployment in .NET. An application is compiled into at least one assembly, which contains its entry point and is stored as an executable .EXE file. The main assembly typically also references several other assemblies, which are stored as .DLL files[2] and contain class libraries. 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 culture[3] 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.

2.1 Assembly Lookup Service

The most central feature of LADY is a globally available and resilient lookup service that, given the strong name of an assembly, can tell you where to find it. LADY can also determine if an assembly has been superseded by a more recent version, and provides the functionality for actually 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 (DeCandia et al., 2007) 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.

For each unique $(name, culture, publicKeyToken)$ tuple we store a *base record* containing the information common across all versions of that particular assembly. Most notably, the base record contains a list of known assembly versions, so that wildcard queries, for example to request the most recent version, can be satisfied. For each specific version of an assembly, we also store an *assembly record*, which is keyed by the full $(name, culture, publicKeyToken, version)$ tuple of the assembly; in other words, its strong name. The assembly record contains information about how to obtain that specific version. For example, the record may contain a download URL, or a NuGet[4] package identifier and version.

2.2 Security

The *public-key token* of an assembly is defined by the .NET framework as an 8-byte hash of the public key

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

[3] A culture is a .NET abstraction for localization.

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

in the key pair that was used to sign the assembly. It is commonly displayed as a 16-digit hexadecimal number. Since the public key token is determined by the signer of the assembly, and also incorporated into its strong name, it is not possible to modify a signed assembly without knowing the private key of its signer or breaking the RSA encryption. With access to the source code, you could recompile the assembly and sign it with another key pair, but that would result in a different strong name.

While this protects against malicious tampering with the code in an assembly, we would also like to guarantee that LADY can provide genuine and valid locations for assemblies. 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 this, 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 perform the registration of an assembly, LADY provides a simple command-line utility. This is suitable for scripting and can be integrated into build systems so that new releases are registered automatically whenever an assembly's version number is bumped. The utility does not directly update the cloud database, and lacks the permission to do so. Instead, it constructs a registration message, signs it with a key pair provided by its user, and sends the signed message to LADY. The service verifies that the message is authentic and that the signer's public key matches the public key token of the assembly, before updating the database. Below is one example usage of the lady command-line client:

```
$ lady register -a MyLibrary.dll -p
   MyLibrary -v 1.2.4 -k mykey.pfx
```

Here, the assembly to register is `MyLibrary.dll`, specified with the `-a` option. LADY uses reflection to extract the strong name of the assembly. 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 1.2.4 of the MyLibrary package. Finally, the `mykey.pfx` file, specified with the `-k` option, contains the key pair of the user. It is used to sign the registration message, and if the public key does not match the public key token of the assembly, the registration will fail.

We have settled initially on this model, where assemblies are registered explicitly by their creators. It would also 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 1. Here, the code calls LoadAssembly at an early point in the program execution[5] 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 application 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, which is a verbose and error-prone task. The verbosity stacks up when multiple types are involved; in the example, the

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

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));
   }
}
```

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.

In sum, all these drawbacks of relying on 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 benefit 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.

Staying with the example in Code 1, the StaticallyTypedAccess method shows this approach in action. The method does the exact same thing as AccessUsingReflection, only with the clarity and safety of normal syntax and 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 spe-

cific 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 also supports loading assemblies as a result of 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 (Microsoft Developer Network, 2016).

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* (sem) 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 (Raemaekers et al., 2014). 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 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 (Microsoft, 2015). 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 ap-

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();
    }
}
```

plication 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 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 its assembly lookup service, LADY adds a level of indirection to the process of loading an assembly. This relieves applications of various responsibilities, and enables several useful applications, but 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 its user-defined map function cannot be located. (LADY does maintain a client-side cache, but it could be missing there, too.)

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

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

large. While cloud databases like DynamoDB generally promise great scalability, this scalability 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, our workload is almost completely read-only. This implies that caching can be an effective way to reduce both load and latency, and DNS is a globally distributed cache 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 globally unique (strong) assembly names with their metadata. To locate an assembly, or check if any new versions are available, a quick DNS lookup will, in practice, suffice in the majority of cases. 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.

For the DNS integration we rely on some of our previous work (Pettersen et al., 2014), which mirrors all database keys as labels in the DNS namespace, translates database lookups into DNS requests on the client-side, and sets up a relay-node 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 (Pettersen et al., 2014).

4 EVALUATION

Many of the reasons to adopt LADY are anecdotal, and it is hard to quantify benefits like flexibility and extensibility. However, there may also be concrete performance benefits, as we will demonstrate in this section. As a case study and vehicle for our evaluation

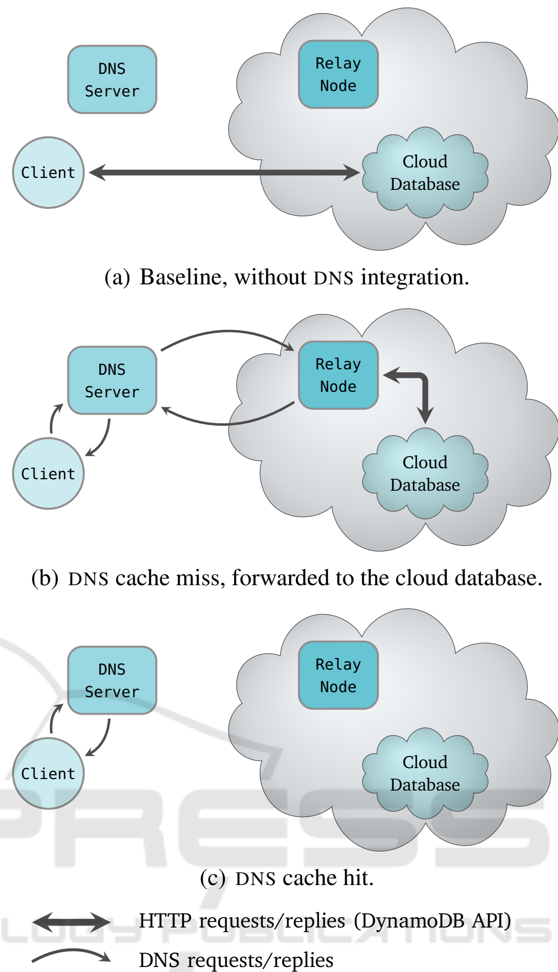

Figure 1: Assembly lookups with and without DNS integration.

we use our previous work on *satellite execution* (Pettersen et al., 2015). This is a technique to reduce latency for applications that need to interact repeatedly with cloud services, by temporarily offloading code so that it executes in closer proximity to the cloud, with lower latency to the targeted services. While exploring this concept we developed a programming abstraction called *mobile functions*, and an *execution server* designed to receive and execute offloaded mobile functions.

In our original implementation, which we will refer to as the baseline implementation, the execution server received mobile functions as serialized objects, which were then deserialized and allowed to execute by invoking an entry point implemented by the objects. Deserialization can fail if an unresolvable assembly reference is encountered. We handled this and other assembly resolution errors by returning an error to the client. The client would then upload the missing assembly to the execution server, before making a

new attempt to offload the mobile function.

This design 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 server may encounter assembly resolution errors and will have to handle them in some way. Both of these assumptions are reasonable, but the resulting design may cause excessive back-and-forth communication between the client and the execution server for mobile functions that depend on multiple assemblies.

When refactoring our implementation to use LADY, we were able to simplify the code both in the execution server and on the client-side, while making the whole system more robust. The refactored execution server can simply deserialize a mobile function and trust LADY to resolve assemblies, *without* interacting with the client. 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.

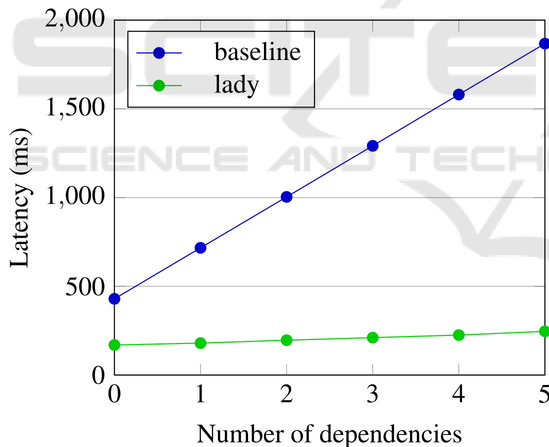

Figure 2: Observed mean latency when executing a mobile function with a varying number of assembly dependencies with and without LADY.

To compare our two implementations, we set up an experiment where the execution server was hosted on an Amazon EC2 node in Ireland, and our client executed from a desktop computer in Norway. Typical ping latency between the client and the server was around 64 milliseconds. We stored a set of assemblies in a separate table in DynamoDB, 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.

Figure 2 shows the difference in latency between our baseline implementation of satellite execution, and the refactored implementation that uses LADY. Given that the motivation behind satellite execution is to reduce latency, these savings in latency are highly significant. Even when the mobile function has no additional dependencies beyond its own assembly, we save a round-trip of communication between Ireland and Norway. By comparison, LADY is only making low-latency DNS requests to resolve assemblies, coupled with lookups to DynamoDB to retrieve the assembly data. So, 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.

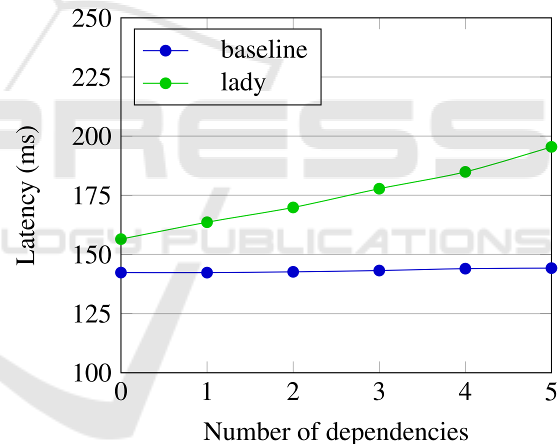

Figure 3: Observed mean latency when executing a mobile function with all assembly data present, where LADY performs DNS lookups to ensure that the latest versions will be loaded, and the baseline does not check for updated versions.

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 a DNS lookup for each assembly that is resolved, in exchange for guaranteeing that the most recent version of the assembly will be loaded. Whether this is a sensible trade-off for an application depends on its 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

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

to the number of dependencies, as each dependency adds another DNS lookup.

It should be noted that this overhead is only significant if latency is a concern. The actual resource requirements for performing a DNS lookup are negligible.

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 (Hertzog and Mas, 2006). 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 (Johansen et al., 2007; Johansen and Johansen, 2008). Systems distributed commercially, like Microsoft Windows, often come equipped with proprietary mechanisms for distributing software updates (Gkantsidis et al., 2006) and are generally less vulnerable to intrusions.

In the framework for code updates described by (Hicks et al., 2001), 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 (Segal and Frieder, 1993). DYMOS (Cook and Lee, 1983) 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 (Gilmore et al., 1997). 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 (Zhang et al., 2014) 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

A key idea underlying LADY is to make all code live in a globally accessible namespace so that it can be referenced unambiguously by name and retrieved on demand in any context. Strong-named .NET assemblies already have globally unique names, but the ability to load code in any context is missing. LADY fills in this gap by creating a lookup service for assemblies, and by implementing the mechanisms for obtaining code on demand. This aligns with a vision where code can be deployed only once, and then instantiated anywhere, in various configurations.

The general approach of loading code on demand means that *distribution of code is decoupled from dis-*

tribution of state. In other words, code does not have to be propagated through a distributed system along the same communication paths as data. Consider, for example, a system where nodes communicate over a gossip-based protocol. A message might contain serialized data and traverse multiple edges of the gossip graph before it arrives at a node where the data must be deserialized. Any intermediate nodes will only be passing along the serialized data and may never have a need for the associated code. But the sender does not know if the target node has the requisite code installed. So to be safe, the sender will have to include the possibly redundant code as part of its outgoing message, or the design must be complicated in some other way, for example by adding additional rounds of gossip to retrieve the code.

With the separation of concerns that LADY offers, the design of such gossip-based systems could be simplified, since code would be retrieved on demand via an entirely independent mechanism whenever data was deserialized. Our satellite execution refactoring in section 4 also helps to illustrate how LADY can simplify the design of other distributed systems, to improve extensibility and serve as a convenient foundation for mobile code.

REFERENCES

Semantic Versioning. http://semver.org/.

Cook, R. P. and Lee, I. (1983). DYMOS: A dynamic modification system. volume 8, pages 201–202, New York, NY, USA. ACM.

Daubert, E., Fouquet, F., Barais, O., Nain, G., Sunye, G., Jezequel, J.-M., Pazat, J.-L., and Morin, B. (2012). 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*, pages 10–11.

Dean, J. and Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th symposium on Operating Systems Design and Implementation*, OSDI '04, pages 137–150. USENIX Association.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220. ACM.

Gilmore, S., Kirli, D., and Walton, C. (1997). Dynamic ML without dynamic types. Technical report, University of Edinburgh.

Gkantsidis, C., Karagiannis, T., Rodriguez, P., and Vojnović, M. (2006). Planet scale software updates. *ACM SIGCOMM Computer Communication Review*, 36(4):423–434.

Hertzog, R. and Mas, R. (2006). *The Debian Administrator's Handbook*. Freexian SARL, https://debian-handbook.info/, first edition.

Hicks, M., Moore, J. T., and Nettles, S. (2001). Dynamic software updating. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 13–23, New York, NY, USA. ACM.

Johansen, D., Lauvset, K. J., van Renesse, R., Schneider, F. B., Sudmann, N. P., and Jacobsen, K. (2001). A TACOMA retrospective. *Software - Practice and Experience*, 32:605–619.

Johansen, H. and Johansen, D. (2008). Resilient software mirroring with untrusted third parties. In *Proceedings of the 1st ACM workshop on hot topics in software upgrades (HotSWUp)*.

Johansen, H., Johansen, D., and van Renesse, R. (2007). Firepatch: secure and time-critical dissemination of software patches. In *Proceedings of the 22nd IFIP International Information Security Conference*, pages 373–384. IFIP.

Microsoft (2015). Application Domains. http://msdn.microsoft.com/en-us/library/cxk374d9%28v=vs.90%29.aspx.

Microsoft Developer Network (2016). *Best Practices for Assembly Loading*. Microsoft, .NET Framework 4.6 and 4.5 edition. https://msdn.microsoft.com/en-us/library/dd153782(v=vs.110).aspx.

Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. (2008). Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110. ACM.

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

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

Raemaekers, S., van Deursen, A., and Visser, J. (2014). 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*, pages 215–224.

Segal, M. and Frieder, O. (1993). On-the-fly program modification: systems for dynamic updating. *Software, IEEE*, 10(2):53–65.

Valvåg, S. V., Johansen, D., and Kvalnes, A. (2013). Cogset: A high performance MapReduce engine. *Concurrency and Computation: Practice and Experience*, 25(1):2–23.

Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Gunda, P. K., and Currey, J. (2008). DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating Sys-*

tems Design and Implementation, OSDI'08, pages 1–14. USENIX Association.

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