Building Dynamic, Long-Running Systems

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI. 02912 USA
spr@cs.brown.edu

Qi Xin
Department of Computer Science
Brown University
Providence, RI 02912 USA
qx5@cs.brown.edu

ABSTRACT

Complex applications that are effectively systems-of-systems are becoming more common and more useful. Our goal is to devise new ways of architecting such systems that will make their programming easier. We take a component oriented approach. A component's interface, which we call an outerface, includes not only the syntax of the component, but also its semantics and constraints on its use. Implementations of outerfaces are defined separately. Our underlying framework, TAIGA, lets the user code directly to the outerface and automatically finds, validates and binds an appropriate implementation. The framework handles component evolution and failure by detected changes and dynamically revalidating and rebinding possibly new implementations to existing outerfaces while maintaining the running system. We are currently working on extending this framework to handle modern, distributed systems-of-systems.

Categories and Subject Descriptors

CCS: Software and its Engineering: Software Organization and Properties: Software System Structure: Ultra-large-scale systems.

Keywords

Distributed systems; evolution; interfaces.

1. INTRODUCTION

Long-running, complex systems-of-systems have been around for a long time but are now becoming increasingly common and less specialized. Originally, systems of systems were designed for closed, stable environments such as telephony, ships, airplanes, or automobiles. Today, common applications, such as Waze [9] can be considered systems-of-systems since they depend on code running on both a wide variety of local devices (e.g. phones) as well as systems running on a distributed set of servers.

Such applications are difficult to write and maintain. They are long running and can't easily be taken down and restarted. They are distributed, with components running on a variety of devices linked by potentially unstable networks. They have widely varying loads over time. They must handle recovery and dynamic evolution. Moreover, as they become more common, security and similar concerns become more important.

Our research is aimed at making it relatively easy to write such large-scale applications that make use of today's computers and environments and that can address these various problems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SESoS'16, May 14-22, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4172-1/16/05#$15.00
DOI: http://dx.doi.org/10.1145/2897829.2897831

2. PROBLEMS

There are a number of issues that have to be addressed when programming such complex, long-running systems.

The systems are first characterized by the use of distributed components running on machines that may be out of the control of the originator. These components are likely to both change and fail. For example, a web service might go down arbitrarily. Over time, services used by the system might evolve, adding additional functionality or even removing existing functionality. (For example, the original Google search service returned the open-directory category for each web page. At some point, this was discontinued although the data field for it remained.) Operating systems, libraries, etc. on user-owned devices will be updated periodically with new or modified functionality. Open source libraries will evolve. Portable devices will be shut down and periodically lose or gain connectivity.

A second issue is that these evolving applications should be able to make use of the data available from today's devices. Waze, for example, uses position updates on phones to deduce traffic conditions. Other applications might want to make similar use of position information, weather information, health information, etc. A new application shouldn't have to duplicate the code to obtain or process this information from multiple devices. Suppose, for example, one wanted to write an application for handling emergencies. One would want to access health information from those in the affected area but it would be unreasonable to expect that everyone was running you application to start with. When providing and accessing such data, one also needs to take into account security and privacy concerns.

A third issue is that dynamic applications should be able to make effective use of dynamically changing computing capabilities. An application using speech recognition might prefer to send the raw data to a server to do the recognition. However, if the server is not available, there might be other source, perhaps a secondary server that is slower or less accurate, that could be used. If this is not available, it might be possible to do limited accuracy recognition on the local device itself. The system should handle such changes

```
outerface edu.brown.cs.webview.taiga.WebManager {

    description {{
        This outerface manages a set of files for the webview application, ensuring
        that they do not get too long.  A transfer record is added to a file when it
        does exceed the 1M length limit
     }}

    trait { rebind=true; }

    class FileManager {
        static public String getCurrentFile();
        static public String getFileForDate(long date);
    }

    testcase Test0 {{
        public static void test() {
            FileManager.getCurrentFile();
            TaigaTesting.success();
        }
     }}

}   // end of outerface WebManager
```

Figure 1. Sample Outerface Definition.

automatically while taking into account system load, power, availability, accuracy, etc.

A fourth issue is that such systems are quickly becoming too big for one person or even one programming team to build everything. To make programming such systems easier, one would want to utilize existing solutions wherever possible. This means embracing open source code, and making use of systems and components (such as web or micro services and libraries) that are written and maintained by others. Such use must take into account such factors as security and costs, and must deal with the fact that these components can and will change without notice.

3. OVERVIEW

In order to address these problems and make programming large-scale open-source, distributed, evolving systems easier, we need new ways of thinking on how they are constructed. A suitable programming framework for such systems should make them straightforward to code. The framework should address both transient and permanent failures. It should handle the evolution of the various component systems. It should handle data sources and access. It should do all this while addressing security and privacy concerns as well as various costs.

An appropriate framework for constructing such complex systems is a hierarchical component-based model. Each of the subsystems of the system can be viewed as a component. These systems are in turn made up of other components, either external such as micro services, or internal such as libraries, classes, or other packages. Failures can then be viewed as component failure; evolution as component evolution; security as component security.

Such a framework puts a heavy emphasis on the interfaces between the components. This is consistent with accepted methodologies for building systems of systems [5] where the underlying systems are just implementations of these interfaces.

The problem that arises in this model is what is a component. The standard notion of a component is that there is a well-defined interface and the component is an implementation of that interface. Then the question is what is an interface.

A typical interface defines the calling sequences for using the component. This is primarily a syntactic definition, providing the calling names, signatures, passed and returned data types, etc. While this is necessary for programming evolving systems, it is not sufficient for a complex application. Interfaces need to include more information and need to include it in a way that it can be used directly by the underlying system and its framework.

The primary information beyond the syntactic is the interface's semantic definition This should describe what the component does and how it should be used. To be used in a practical framework, this definition needs to be understandable to machines. Formal specifications, while meeting this requirement, are generally not practical to write for large scale components and often cannot be written at all. Other techniques, such as test cases and partial contracts, are more practical and useful.

Beyond semantic information, the interface needs to include other data that will be needed by the framework for handling evolution, security, and similar constraints. The interface should include a security model describing what the implementation can and can't do. It should include a cost model that would let the application or framework choose among multiple implementations. It should include information on how to recover or restart on failure. It should include data privacy constraints if private data is involved.

4. PREVIOUS WORK

Our previous work in this area, TAIGA [6-8], was aimed at a slightly different problem, although there is much in common. TAIGA considered desktop applications that made use of web services and similar open implementations. It assumed a crowd-sourced environment where people would contribute designated implementations (e.g. web services, libraries, servers) that would be automatically incorporated into a long-running system with appropriate failure semantics. It did not address any problems involving data.

TAIGA uses a hierarchical component model. Components are defined by *outerfaces*. A sample outerface is shown in Figure 1 An outerface defines all the relevant properties of the component. It includes a Java-like interface defining the syntax of the component. As with Java interfaces, this can include method calls and internal classes and interfaces. Unlike Java interfaces, but

consistent with the overall notion of a component, it can include both constructors and static methods.

Next the outerface includes the semantics of the component. This can be defined either in terms of test cases that any implementation needs to pass, in terms of contracts both for the component and for each method, or a combination of the two. The system guarantees that an implementation passes all the given test cases and is consistent with the contracts in doing so before allowing it to be used. It is possible to restrict an outerface by adding additional test cases or to extend it by adding additional methods.

Finally the outerface includes constraints on the implementation including a cost model, security model, and recovery model. The cost model provides a means for choosing among implementations for this outerface. The security model is based on the Java Security framework [4]. The recovery model provides information for restarting an implementation based on previous calls.

Implementations are defined as mappings from an actual implementation to an outerface. The mapping may specify a set of classes that are to be bound into the application as a library, a set of classes that are to be run independently as a server using a SOAP-based remote procedure call protocol, or as a web service with a particular URL and binding. Implementations can define minor mappings of arguments, names, types, etc. between the actual implementation and the interface defined by the outerface. Implementations can also define additional properties such as the scope of availability and the cost.

The system automatically and dynamically binds implementations to outerfaces. It creates a stub class representing the outerface. The first time a method in the class is called, the system finds an implementation, validates it, and then binds it. Validating the interface involves running the test cases. Choosing among available interfaces is done using the cost model. The binding is done to ensure the security constraints are met whenever possible.

If the component fails, it can be rebound. Failure is detected in lost communication with remote servers, by missing responses from web services, by unexpected exceptions, or by a violation of the contracts imposed by an outerface. Rebinding involves first unbinding the existing implementation and then finding and validating a new implementation. To handle some non-trivial rebindings, the system provides a simple framework for rebinding implementations that maintain a context. The outerface can include a set of variables defining the context and then define how this context changes based on each call. The implementation then must include a constructor taking the context variables as parameters which is then used for rebinding.

The cost model can take into account the time or memory used in the test cases, the type of binding, and the cost of the implementation. It lets the user define a linear combination of these and chooses the implementation with the lowest cost.

The security model provides a context for running the implementation. For code that is bound into the user's application, calls are done through a security portal that imposes the given context around the call. For code that is run on a server and where the server is started by the system, the system will actually start a security sandbox with the appropriate parameters to run the server.

TAIGA is designed to work in a crowd-sourced manner, with users defining outerfaces as well as explicit implementations to previously defined outerfaces. It is supported by a peer-to-peer network that supports searching for outerfaces and implementations, keeps track of available servers bound to outerfaces, and provides global services such as a distributed file system and point-to-point sockets. The overall system works by having a small kernel run on each machine and having local programs talk directly to the kernel which in turn talks to the peer-to-peer network.

While TAIGA is a working system with a number of simple applications and provides a model for a practical system-of-systems framework, it is not at the stage where it can be used as the basis for such a framework. There are several features that are needed and that we are currently working on. These include:

- A means for handling data. Outerfaces to date are procedural and pull-oriented. A data framework for modern systems should allow data services to be defined with push semantics and should provide standard data processing services within the system. Our proposal is describe in Section 5.
- Better security. In order to ensure security and privacy, especially when data is involved, the system has to be secure. This means that the kernel needs to be self-validating, that there needs to be a consistent notion of user identity throughout the system, and that security needs to be validated beyond the simple sandboxes that are only run some of the time.
- Better semantic definition of components. Test cases and partial contracts are much more convenient and more widely applicable than formal mathematical specifications. However, it can be difficult or impossible to define test cases for many of the target components. Other means are going to be needed.
- Support for a wider range of applications including RESTful web services. It should be easy to use these as implementations and they should be integrated into the recovery model.
- Support for distributed access to a database.
- An extended cost model. The current cost model can be thought of as a place holder. The model needs to take into account other factors such as accuracy and power consumption. It also needs to be dynamic, since the factors that affect the choice of implementation can change over time. In addition, the cost model should be extended to starting servers on a grid. TAIGA currently just polls the local network to find a machine on which to run a server.
- Deploying at scale. TAIGA has been run with up to 100 nodes, but not at the much larger scale needed for today's mobile-based systems of systems. Moreover, the current system is oriented toward workstations (i.e. almost always connected, no concerns regarding power, etc.).

Our current work involves extending and redoing the existing TAIGA implementation so that it is suitable for implementing systems of systems, addressing these and other problems.

5. DATA PROCESSING

Many of the new potential systems-of-systems involve collecting and using data. For example, a mapping applications might collect data from phones regarding speeds and positions and use it to derive road conditions. A home monitoring service could check the health of various devices and report any problems. An emergency management system could look for health monitors on people in the affected area and flag any problems.

```
dataface edu.brown.cs.taiga.location.GPSData {

    description {{ location and speed information }}

    location { double latitude, double longitude };
    double accuracy;
    double speed default 0;
    double heading;

    restricts {
        -90 <= latitude <= 90,
        -180 < longitude <= 180,
        0 <= speed < 10000,
        0 <= heading < 360
    }

    units {
        accuracy : meters;
        latitude : degrees;
        longitude : degrees;
        heading : degrees;
        speed: meters / second;
    }

    aggregations {
        location : within;
        timestamp : within;
        speed : within, mean, median, mode, variance;
    }

    filters {
        location { * }
        speed { $ <= speed < $ }
        timestamp { $ <= timestamp < $ }
    }

}   // end of dataface GPSData
```

Figure 2. Sample Dataface

Writing an application that makes use of such data today involves creating the back ends that process the information, multiple front ends (one for each type of applicable device) to generate the appropriate data, and all the code needed to connect them. Much of this effort could be shared among applications. For example, position and movement data could be used for crowd control, accumulating historic data on traffic patterns, or identifying hiking or biking trails. Similarly, the type of processing required to aggregate the data or to compute speed from changing positions could be reused. Moreover, any data collection scheme has to include code to make it robust, to handle failure, and to handle data sources that come and go. This code should not have to be duplicated for each application. TAIGA provided outerfaces and their implementations to handle reuse and the dynamics of callable code. A similar mechanism can be used to handle data.

We propose that data be viewed as another type of component. In particular we define data the might be desired by an application using a complete interface we call a *dataface*. A dataface is similar to an outerface except for data. It defines the desired data and provides a standard way of accessing it for an application. The application can be coded directly to the database without regarding the actual implementation. Using datafaces, the various mapping applications noted above could use a common dataface that describes location data. They could then be coded as it that data were readily available and robust.

Similar to an outerface, a dataface includes syntax, semantics, and other considerations. The syntactic definition defines the fields, data types, and the set of operations that can be done on the data. Operations would include different types of aggregations and filters. The semantic definition in a dataface defines the expected units for data, consistency properties, and any data constraints. It lets the system determine if the provided data is valid and sensible. The other considerations would include the cost of obtaining the data, security and privacy considerations, the geographical domain where the data is available, and third-party validation mechanisms to ensures the validity of the requester.

An example dataface is shown in Figure 2. The top line defines the dataface while the next provides a readable description that can be used to understand what it does. The next set of lines defines the fields of the dataface. These are similar to fields in a structure or columns in a database table. The next set of lines provides restrictions on the values of the fields. These provide consistency and range checks. The following units declaration provides the unit type for each of the fields. The system will have a built in set of known units for which it can do automatic conversions. The next two sections define the valid aggregation and filter operations. The *within* aggregation is a bucketing operation that takes a value and forms uniform buckets of that size. A filter type of * allows any filtering operation, while a specific filter expression restricts filters to that form. Note that these refer to the field “timestamp”. This field is created automatically for all datafaces and contains the time the data was generated.

A dataface can have multiple implementations or data providers. A data provider defines how data can be accessed from a particular source. It provides code to build the resultant dataface structure from available resources (although the code could be specified non-procedurally). It provides additional privacy and security constraints on the use of the data, for example limiting the use of the data to aggregates of at least a certain size or limiting the data to authorized applications. It provides its own constraints and consistency properties that can be matched to the dataface. The provider also indicates whether the data is pushed or pulled.

```
dataprovider edu.brown.cs.taiga.location.WebGPS {

   description {{ information from html5 geolocation }}

   implements edu.brown.cs.taiga.location.GPSData;

   using edu.brown.cs.taiga.weblocator.WebGPS;

   restricts {
      -90 <= latitude <= 90,
      -180 < longitude <= 180,
      0 <= speed < 10000,
      0 <= heading < 360
   }

   units {
      accuracy : meters,
      latitude : degrees,
      longitude : degrees,
      heading : degrees,
      speed: meters / second
   }

   trait {
      min_aggregate = 10
   }

}  // end of data provider WebGPS
```

Figure 3. Sample Data Provider Definition

```
SELECT count, location, timestamp, FROM GPSData
AGGREGATE location { latitude within 0.001, longitude within 0.001 },
FILTER accuracy < 50
WINDOW 30000


SELECT location, timestamp, speed, heading FROM GPSData
FILTER accuracy < 10
WINDOW 5000
```

Figure 4. Sample Dataface Queries

An example data provider definition for the dataface of Figure 2 is shown in Figure 3. The implementation clause identifies the dataface that is being provided; the using clause specifies where the code is for the provider. To be used, the restricts clause must be consistent with the definition in the dataface. Similarly, the units clause must be consistent, although automatic conversion of known units is allowed. The defined trait would ensure that any aggregate of fewer than ten data elements would be discarded for privacy purposes.

In order to access the data, the application calls static methods that are generated automatically for the dataface. We expect two types of calls. The first gathers data on demand according to a particular filter and aggregation policy. Such a request would be pushed to all relevant providers. The internal peer-to-peer network would be responsible for aggregating the data and returning the summary to the caller. This would off load the data processing from the application and client and would let applications share the gathered data. The second call is a stream-based query. It would also specify filters and an aggregation policy and would include a time frame and a callback routine. Data would be continuously gathered (either by the providers offering the data to the system or by the system polling providers), filtered, and aggregated in the network and the results would periodically be sent back to the caller using the callback routine. To make this simpler from the programmer's perspective we are considering various SQL stream-based query languages [1,2],

Two possible queries or requests for the dataface of Figure 2 are shown in Figure 4. Both refer explicitly to the dataface and not to the implementations. The first would return (via an appropriate callback) aggregations of location data every 30 seconds. The aggregation would be in buckets of 0.001 degrees latitude and 0.001 degrees longitude and would only include values where the error is less than 50 meters. The second would return all data points with location, speed and heading every 5 seconds provided that the accuracy is less than 10 meters. Note that the example data provider would not offer such information since the minimum aggregation size would not be met.

Security and privacy are a major concerns when accessing data that may be sensitive or privileged. We plan to address these concerns in various ways. First, the data providers will be able to limit access to the data based on the filters involved and a minimum aggregation count. While this doesn't provide absolute protection, it would make it much more difficult to identify individual data elements. Second, we plan to have data that can be released only upon valid authorization of the requester. This would allow the data providers to be selective in terms of who can access the data. Finally, we are looking into applying differential privacy

techniques in the processing of data to provide approximate results while ensuring privacy [3].

6. STATUS

TAIGA is a working prototype. It is running at Brown on a variety of machines and is accessible from outside. In addition to the work on data processing, we are currently extending the implementation to handle a variety of platforms (e.g. phones, Internet of Things), improving the underlying network, improving security, enhancing the testing framework to handle a wider variety of tests, extending the cost model to handle power and accuracy, applying the cost model to finding grid nodes to run servers on, adding RESTful implementations, and developing new sample applications.

Source code for TAIGA is available from our web site (http://www.cs.brown.edu/people/spr/research/taiga.html) or via ftp at ftp://ftp.cs.brown.edu/u/spr/taiga.tar.gz.

7. ACKNOWLEDGMENTS

This work is supported by the National Science Foundation grant CCF1130822.

8. REFERENCES

1.

2. Arvind Arasu, Shivnath Babu, and Jennifer Widom, "The CQL continuous query language: semantic foundations and query execution," *The VLDB Journal* **15**(2) pp. 121-142 (2006).

3. Cynthia Dwork, "Differential privacy: a survey of results," pp. 1-19 in *Proceedings of the 5th international conference on Theory and applications of models of computation*, (2008).

4. J. Steven Fritzinger and Marianne Mueller, "Java Security," *Sun Microsystems*, (1996).

5. M. W. Maier, "Architecting principles for systems-of-systems.," *Systems Engineering* **1**(4) pp. 267-284 (1998).

6. Steven P. Reiss, "Evolving Evolution," *8th International Workshop on the Principles of Software Evolution*, pp. 136-139 (September 2005).

7. Steven P. Reiss, "A component model for Internet-scale applications," *Proceedings ASE 2005*, pp. 34-43 (November 2005).

8. Steven P. Reiss, "Designing Internet-based software," *Proceeding of the Second International Conference on Design Science Research*, (May 2007).

9. Waze, "Waze Home Page," *http://www.waze.com*, (2016).