Reusable and Extensible Fault Tolerance for RESTful Applications

John Edstrom and Eli Tilevich
Virginia Tech
{*edstromj,tilevich*}*@cs.vt.edu*

***Abstract*—Despite the simplicity and scalability benefits of REST, rendering RESTful web applications fault-tolerant requires that the programmer write vast amounts of non-trivial, ad-hoc code. Network volatility, HTTP server errors, service outages—all require custom fault handling code, whose effective implementation requires considerable programming expertise and effort. To provide a systematic and principled approach to handling faults in RESTful applications, we present FT-REST—an architectural framework for specifying fault tolerance functionality declaratively and then translating these specifications into platform-specific code. FT-REST encapsulates fault tolerance strategies in XML-based specifications and compiles them to modules that reify the requisite fault tolerance. To validate our approach, we have applied FT-REST to enhance several realistic RESTful applications to withstand the faults described in their FT-REST specifications. As REST is said to apply verbs (HTTP commands) to nouns (URIs), FT-REST enhances this conceptual model with adverbs that render REST reliable via reusable and extensible fault tolerance.**

***Keywords*-fault tolerance, web services, REST, software reusability, software extensibility**

I. Introduction

As computation is moving from the desktop into the cloud, an increasing number of applications rely on remote functionality provided by web services. Although traditional web services follow the SOAP standard [1], Representational State Transfer (REST) [2] has been making deep inroads into the design of the next generation of cloud-based applications. As compared to the previous generation of SOAP-based applications, RESTful applications offer simplicity, scalability, and composability advantages. Coined by Roy Fielding in his Ph.D. dissertation, the REST architecture codifies a set of principles for constructing network-based software, in which a small set of commands manipulate heterogeneous network resources. The canonical example of REST is HTTP, a ubiquitous network protocol that manipulates uniform resource identifiers (URIs) by means of four primary commands: PUT, GET, POST, and DELETE. A foundation of the world wide web, REST is increasingly becoming the preferred standard for defining web services. In the context of this paper, we refer to *RESTful web services* as those made available over HTTP.

RESTful services have seen enormous growth and adoption in recent years. Consider the Programmable Web [3], a web services directory that counts more than 3,000 REST APIs as of this writing. Furthermore, numerous programming frameworks have been developed to facilitate accessing RESTful services from mainstream programming languages. Examples of popular commercial frameworks for accessing RESTful services include the Apache HttpClient [4] for Java and the httplib [5] for Python. In addition, server-side programming frameworks, including Jersey [6] for Java and Ruby on Rails [7], provide facilities for easy development and deployment of RESTful services. By alleviating the burdens of developing RESTful applications, these frameworks spur the growth of RESTful applications.

RESTful applications, as all distributed systems, are vulnerable to partial-failure, in which different parts of a distributed execution can fail independently. Unfortunately, REST programming frameworks are designed under the assumption that fault-tolerance is highly application specific, and as such is better left for the programmer to implement. Although this design assumption is true in general, the thesis of this work is that fault tolerance in RESTful applications can be rendered reusable and extensible through an expressive programming framework.

In particular, we observe that the prevailing majority of faults in RESTful applications are due to one of the following three conditions: network volatility, service unavailability, and server errors. Furthermore, each of these conditions can be reliably detected with straightforward system implementation idioms. For example, network volatility is exemplified by unusually high latencies, congestion, dropped packets, and low bandwidth—all reported as exceptions in the network APIs of modern programming languages. Service unavailability may not be trivial to detect, but if this condition is not handled effectively it can quickly render the underlying application unusable, particularly if the application comprises multiple, distinct services. Nevertheless, a simple systematic policy that, for example, times-out after a given threshold can meaningfully inform the user who can then take corrective actions. HTTP server errors have been standardized as error codes that can easily be identified and appropriately mitigated.

The existing state of the art in creating robust RESTful applications require that the programmer code defensively taking into consideration all the possible conditions that may arise during the execution of the application. These conditions depend not only on the application's business logic, but also on the specificities of the application's deployment environment and operation mode. As a result, the code required to render a RESTful application fault tolerant can quickly grow in complexity because of the need to handle various special cases, specific to different deployments. What is even worse is that this painstakingly developed fault tolerance code is inherently non-reusable, as each deployment environment and application may possess a unique combination of fault characteristics. Plagued by the necessity to provide custom fault handling code, the RESTful programmer is left unable to leverage any uniformity in the fault characteristics across services.

In this paper, we propose a new approach to rendering RESTful applications fault tolerant that exploits their architectural commonalities to provide reusable and extensible fault tolerance modules. Our approach—called FT-REST—is an architectural framework realized as a domain-specific Fault Tolerance Description Language (FTDL) and a client-side library. To add fault tolerance to a RESTful application, programmers specify the fault tolerance policies in FTDL; the FT-REST framework then compiles these FTDL specifications into platform-specific code modules that can be added to the business logic, rendering it fault tolerant. FTDL specifications can be reused not only across applications, but can be compiled to a variety of platforms. Furthermore, existing FTDL specifications can serve as convenient building blocks for designing custom fault handling strategies.

By addressing the limitations of existing approaches to fault tolerance in RESTful applications, this paper makes the following contributions:

- **Fault Tolerant REST (FT-REST)**: an architectural framework that systematically enhances RESTful applications with platform-specific fault tolerance functionality.
- **Fault Tolerance Description Language (FTDL)**: an XML-based domain-specific, platform-independent language that enables reusable and extensible fault tolerance for RESTful applications.
- **Empirical evaluation**: a set of case studies that demonstrate how FT-REST can render realistic RESTful applications resilient against faults common to the REST architecture.

The rest of this paper is structured as follows. Section II motivates the problem. Section III outlines major fault tolerance strategies in the literature. Section IV presents the FT-REST framework and the Fault Tolerance Description Language (FTDL). Section V evaluates our work through case studies. Section VI discusses the potential impact of FT-REST. Section VII compares this work to the related state-of-the-art, and Section VIII presents concluding remarks.

II. Motivation

A typical e-commerce service enables remote users to browse product catalogs, place orders, and track shipments. These functionalities are commonly exposed as a RESTful web service interface. Depending on the relative locations of the service's clients, they may experience various kinds of faults when invoking different service operations, including network volatility, temporary service unavailability, and server errors. To handle these faults, programmers employing this service must provide custom fault handling functionality that suffers from three major limitations: programmability, reusability, and extensibility.

```java
HttpClient cli = new DefaultHttpClient();
cli.getParams().setIntParameter(
  CoreConnectionPNames.SO_TIMEOUT, 5000);
HttpResponse resp = null;
for (int i = 0 ; i < 5 ; i++) {
  try {
    resp = cli.execute(
        new HttpGet("http://example.org/products"));
  }
  catch (ClientProtocolException e) {
    //Handle exception
  }
  catch (IOException e) {
    //Means we timed out. Sleep then retry.
    try {
      Thread.sleep(1000);
    }
    catch (InterruptedException ex) {
      //Handle exception
    }
    continue;
  }
  if (resp != null)
    break;
}
```

Figure 1. Making a GET request using Apache's HTTPClient library. Retrying five times at intervals of 1000 ms.

A. Programmability

In a typical client application, fault tolerance functionality is cumbersome to implement. Consider using the Java Apache HttpClient [4] to obtain a list of products (Figure 1). The fault tolerance functionality is simple: if the network times-out, sleep for one second and retry; repeat five times before giving up. However, the code to implement this simple fault tolerance functionality is scattered throughout the method, located in exception handling clauses. A maintenance programmer would be challenged to understand how this code handles faults.

Notice that almost every RESTful application will likely need to put in place a fault handling policy as presented here. And yet, this policy is scattered around 25 lines of code. The implemented policy has hard-coded values for the timeout duration, number of retries, and the idle period between the retries. Also, this is somewhat of a toy example, because typically invoking a service requires that more than one fault is properly detected and handled. It would not be unusual to expect a service programmer to spend more time implementing various fault tolerance features than implementing the core service functionality.

B. Reusability

The only way to reuse the fault tolerance functionality in Figure 1 is through copy-and-paste, which is prone to introducing insidious errors. Additionally, all variables associated with the fault tolerance functionality would have to be properly updated for each new service invocation. But what it is worse is that any new fault characteristics introduced by a service must be handled in concert with the fault handling functionality.

With respect to fault tolerance, reusability can be achieved at three different levels. First, intra-application reusability enables all faults pertaining to invoking the same service in a given application to be handled uniformly. In our example, an order-placement service invoked from multiple locations within an application should be able to reuse the same fault tolerance functionality. With existing fault tolerance implementation mechanisms, intra-application reusability is impossible.

Second, inter-application reusability enables all faults pertaining to invoking the same service across multiple applications to be handled uniformly. In our example, the product browsing service will likely be used by multiple applications. Although each application will use this service for its individual business purposes, the potential faults are likely to be identical, defined by the nature of the service rather than the business context under which it is used. However, the hand-coded fault tolerance functionality does not lend itself to easy reuse even within the ecosystem of a single language.

Finally, cross-platform reusability enables all faults pertaining to invoking the same service across multiple applications written in different languages to be handled uniformly. Standard web services have achieved a remarkable level of cross-platform reusability of their core logic, in which XML-based interfaces can be easily translated to any client platform. Unfortunately, the corresponding manually-coded, fault tolerance functionality has to be reimplemented for each client platform. As a specific example, the code within Java exception handlers would not be reusable in C#, even if the fault characteristics of the service and the fault tolerance policy in place are exactly the same. When the amount of custom fault handling code starts to surpass that of core business logic, the Platform Lock-In anti-pattern [8] is likely to manifest itself.

In summary, traditional approaches to implementing fault tolerance in RESTful applications cannot achieve reusability at the intra-application, inter-application, or cross-platform levels.

C. Extensibility

Extensibility refers to the programmer effort required to derive closely-related functionality from a given base form. Custom, hand-coded fault tolerance does not lend itself to being easily extended. Consider needing to slightly modify the fault handling code in Figure 1, in response to changes in the deployment environment. Assume that the error code 503 now signals that the web service is temporarily unavailable and has to be reinvoked later. In addition, the wait time between invocations should be now doubled. Furthermore, the number of retries should be increased from five to ten. Finally, the timeout should now decrease to 4 seconds. Figure 2 depicts the code that the programmer must now implement to meet these requirements. Even for a toy example like this, the amount of changes in different locations of the codebase is staggering. As a result, this re-engineering effort is cumbersome and error-prone.

```java
HttpClient cli = new DefaultHttpClient();
cli.getParams().setIntParameter(
  CoreConnectionPNames.SO_TIMEOUT, 4000);
HttpResponse resp = null;
int sleepTime = 1000;
for (int i = 0 ; i < 10 ; i++) {
  try {
    resp = cli.execute(
        new HttpGet("http://example.org/products"));
  }
  catch (ClientProtocolException e) {
    //Handle exception
  }
  catch (IOException e) {
    //Means we timed out. Sleep then retry.
    try {
      Thread.sleep(sleepTime);
      sleepTime *= 2;
    }
    catch (InterruptedException ex) {
      //Handle exception
    }
    continue;
  }
  if (resp != null) {
    if (resp.getStatusLine().getStatusCode() == 503)
      continue;
    break;
  }
}
```

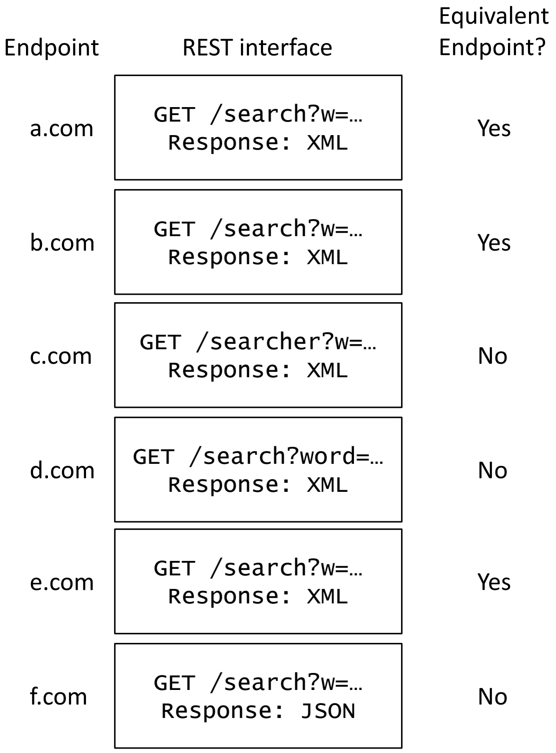
Figure 2. If a network timeout occurs or a status code of 503 is returned, retry the service.

III. Identifying Fault Tolerance Strategies

The key insight that enables our approach is that fault tolerance functionality of RESTful applications follows well-defined patterns. It is by organizing fault tolerance functionality around these patterns that our approach enables higher degrees reusability and extensibility than the existing state of the art.

To identify fault tolerance patterns, we have examined several representative prior approaches to making services more reliable. These approaches range from replicating SOAP web services [9], [10], [11] to introducing transactional processing [12], as well as adding special fault tolerance code to server and client sides of a service-oriented application [13], [14]. Although our approach focuses on client-side fault tolerance, we were able to apply lessons learned from all these approaches when designing FT-REST.

To handle a fault, it must first be detected. The faults in RESTful applications that we aim at handling stem from three classes of abnormalities: network volatility, service



Endpoint	REST interface	Equivalent Endpoint?

a.com	`GET /search?w=…`
	`Response: XML`
		Yes

b.com	`GET /search?w=…`
	`Response: XML`
		Yes

c.com	`GET /searcher?w=…`
	`Response: XML`
		No

d.com	`GET /search?word=…`
	`Response: XML`
		No

e.com	`GET /search?w=…`
	`Response: XML`
		Yes

f.com	`GET /search?w=…`
	`Response: JSON`
		No

Figure 3. Equivalent and non-equivalent service endpoints. In this case, equivalent endpoints must have the URI "/search" that takes one parameter "w" and must return XML.

outages, and internal service errors. Setting and detecting a timeout can usually reliably detect the first two conditions. The challenge is to choose a timeout value that is meaningful for the application at hand and its operating environment. Therefore, timeouts must be easy to specify and fine-tune as necessary. HTTP was designed to convey server-side errors as numerical status codes. In particular, the codes with values greater than 200 indicate a deviation from a normal execution.

Another design consideration is the notion of *equivalent* service endpoints. A web service can be accessed through one of its endpoints, a set of unique network addresses exposing identical service interfaces. When making a RESTful application fault tolerant, the endpoints of its constituent services can be used for implementing various replication-based fault handling strategies. Endpoints are equivalent if their service parameters and responses match, given the same set of parameters. Figure 3 details equivalent and non-equivalent service endpoints.

Having reviewed the research literature on the topics related to fault tolerance in web applications, we have distilled the massive body of approaches and techniques into the following list that we consider relevant and useful in the context of RESTful applications.

A. Retry

Arguably, the most widely used fault tolerance strategy for web services is **Retry**. This strategy prescribes reattempting a service's endpoint in response to a failure for a given number of times. Several parameters determine how a retry is executed: n_r specifies the number of attempts, i backoff interval, and t backoff type (constant, linear, exponential, etc.). Thus, the tuple of $\{n_r, t, i\}$ can express an unlimited number of parameterizations for this strategy.

In fact, the example in Figure 1 implements the **Retry** strategy. Each subsequent iteration through the for-loop executes a GET request to the same service endpoint. In this case, $n_r = 5$ since the number of retries is five, $i = 1000$ since we wait 1000 ms between requests, and $t = constant$ since we are not changing the backoff interval.

On their website, Twitter [15] recommends both linear backoff and exponential for failures in their Streaming API. If the error is related to a network failure (e.g., a timeout), clients should backoff linearly starting at 250 ms and going up to 16 seconds. If an HTTP failure occurs, as signaled by a status code greater than 200, the client should backoff exponentially starting at 10 seconds and ending at 2 minutes. This real world web service is trying to impose a specific fault tolerance strategy on its users. However, programmers are free to handle faults in a fashion that best satisfies their application design needs.

B. Sequential

Another widely used fault tolerance strategy is **Sequential**. Sometimes referred to as *passive* replication, this strategy prescribes that a web service invocation iterates through its endpoints, in response to a failure.

For instance, consider invoking the service endpoint at `a.com/search`, which times out due to high network volume. This strategy prescribes that, when the failure is detected, the endpoint at `b.com/search` is attempted. Although not guaranteed to succeed, this strategy provides yet another opportunity to successfully invoke a given service. This strategy is dubbed *passive* replication because it waits for a failure to occur before falling back to additional endpoints.

C. Parallel

A more complex fault tolerance strategy is **Parallel**. This strategy prescribes that a service be *actively* replicated, with web service endpoints being invoked simultaneously to protect against any potential service unavailability.

For instance, in our example, both `a.com/search` and `b.com/search` can be invoked simultaneously to proactively guard against any of these endpoints being unavailable. This strategy incurs a slight performance overhead by launching two network requests, which use expensive I/O operations. Because this strategy entails speculative parallel execution, the first successfully executed request enables the application to proceed. As a result, the overall performance may improve by exploiting the fastest available network connection. Alternatively, rather than selecting the first result returned, one could implement a strategy whereby the returned services were voted on and the "best" response was selected as described in reference [16].

D. Composite

In practice, a single strategy may not yield the requisite levels of reliability and performance. As a result, system designers often resort to combining multiple fault tolerance strategies. Specifically, all of the aforementioned strategies can be combined into composite fault tolerance strategies. The selection of which strategy to employ is based on a service's QoS characteristics, as highlighted in reference [14].

- **Retry-Sequential:** A group of endpoints will be invoked one after the other. If all of them fail, the entire sequential block is retried.
- **Retry-Parallel:** A group of endpoints will be invoked in parallel. If all of them fail, the entire block is retried in parallel.
- **Sequential-Retry:** A group of endpoints will be invoked one after another, but each endpoint may have its own retry semantics.
- **Sequential-Parallel:** The list of endpoints for a service are grouped into parallel blocks that will be invoked simultaneously. These parallel blocks are then attempted one after another if a failure occurs.
- **Parallel-Retry:** A group of endpoints will be invoked concurrently, but each endpoint may have its own retry semantics.
- **Parallel-Sequential:** The list of endpoints are grouped into sequential blocks that will be invoked one after another. These sequential blocks are then attempted concurrently.

IV. FAULT TOLERANT REST (FT-REST)

Next we present Fault Tolerant REST (FT-REST), our architectural framework that can render any standard RESTful application resilient against specified faults. FT-REST takes advantage of the common fault tolerance patterns found in the implementations of the majority of realistic RESTful applications. FT-REST comprises the following three key architectural components:

1) **Service:** The RESTful service being rendered fault tolerant, defined by its set of equivalent endpoints.
2) **Fault Conditions:** The unique system state that signals the presence of a fault to be handled.
3) **Handling Strategy:** The specific fault tolerant strategy to be followed when handling a given fault (e.g., **Retry**, **Sequential-Retry**, **Retry-Parallel**, etc.).

```
1  <?xml version="1.0" encoding="UTF-8">
2  <!DOCTYPE ftdl SYSTEM "ftdl.dtd">
3  <ftdl>
4    <strategies>
5      <strategy>
6        <service matchesUri="http://example.org/products"
7            method="GET" />
8        <conditions>
9          <timeout>4000</timeout>
10         <status>503</status>
11       </conditions>
12       <sequential>
13         <endpoint uri="http://example.org/products"
14             method="GET" numRetries="10"
15             backoffInterval="1000" backoffType="exponential" />
16       </sequential>
17     </strategy>
18   </strategies>
19 </ftdl>
```

Figure 4. An FTDL specification file

A. Motivating Example Revisited

Consider how FT-REST can be used to render our motivating example in Figure 2 fault tolerant. The FT-REST architectural components for this example correspond to:

1) **Service:** The service consists of only one endpoint located at `example.org/products`.
2) **Fault Conditions:** The two fault conditions that signal the presence of a fault that we want to handle are: (1) a network timeout of 4000 milliseconds, and (2) a status code of 503.
3) **Handling Strategy:** The strategy we want to put in place is **Retry**; this strategy is parameterized with: # retries (10), a backoff interval (1sec), and a backoff type (exponential).

In line with web services standards, we express this FT-REST specification as a machine-readable XML file depicted in Figure 4. We call the XML-based language used for defining FT-REST specifications Fault Tolerance Description Language (FTDL).

An FTDL specification file can define multiple strategies. As seen on line 6, each strategy is delineated by a `<service>` tag, which also contains the URI containing the service being rendered fault tolerant. As stated previously, because a service comprises a set of equivalent endpoints, there may not be a one-to-one correspondence between a service and a URI. For greater genericity, FTDL takes advantage of regular expressions to express multiple endpoints concisely. By specifying a single string, a programmer can tie a web service to its endpoint URI. This design facet of FTDL is well-aligned with one of the foundational principles of REST: resources are to be keyed by unique URIs. In the context of our example, this design facet can be realized as follows. If `http://example.org/products` had backup a endpoint at `http://example.co.uk/products`, the programmer could use `http://example.*/service` to match all endpoints.

One expressiveness benefit of this approach is that the `matchesUri` attribute does not even need to be a well-formed HTTP URI to specify a service. The only requirement is that this identifier be unique amongst other services. In our example, the service could be identified by `ftrest://service1` or even `service1`. Thus, our approach provides a high degree of flexibility with respect to naming conventions that the programmer wants to follow in a given application. The only drawback is that service URIs that are too generic may lead to name conflicts amongst services.

Another advantage of FTDL design is that the list of endpoints need not be explicitly declared with the service. Rather, the endpoints are defined within the strategy's context, as seen on line 13. As a consequence, programmers needing to define complex strategies (e.g., **Sequential-Parallel**) can freely divide the endpoints into blocks as they see logically fit. Instead of hard-coding any particular scheme for splitting the endpoints into groups, our approach provides the programmer with complete flexibility in endpoint assignment.

As seen on line 8, failure conditions are specified in their own `<conditions>` tag. The individual conditions are connected with a logical-or operator that evaluates each condition in sequence. This feature enables programmers to specify multiple conditions that may trigger a fault tolerance strategy. In our current implementation, we support `<timeout>`, measured in milliseconds, and `<status>`. However, the list of conditions can be easily extended.

In this example, the employed strategy is **Retry**. The `<sequential>` tag on line 12 seems to indicate passive backup, although no secondary endpoint is defined. In fact, retrying a single endpoint is a special case of the **Sequential-Retry** strategy, in which the number of endpoints is exactly one. The FTDL compiler enforces the semantics of the root strategy tag taking either the `<sequential>` or `<parallel>` values, as a means of maintaining consistency in the strategy definition. Intuitively, the fault tolerance can either be applied sequentially or in parallel.

The **Retry** strategy's parameters are expressed as XML attributes of the `<endpoint>` tag. By setting these attributes to different values, the programmer can flexibly fine-tune fault handling on a per-endpoint basis. To highlight the versatility of FTDL, consider what it would take to change the fault handling strategy in place from **Retry** to **Retry-Sequential**. The only editing required to put this change into effect is moving the retry attributes to the `<sequential>` tag. Retry attributes can decorate `<sequential>`, `<parallel>`, and `<endpoint>` tags, depending on the strategy.

Once the fault tolerance functionality is encapsulated within an FTDL specification, the main source code file can be streamlined as follows:

```java
FTHttpClient cli = new FTHttpClient();
HttpResponse resp =
  cli.execute(new HttpGet("http://example.org/service"));
```

The only change to the Java API is to replace the Apache HTTP client with our library class, FTHttpClient, that mirrors the method interface of the original Apache class. All the boilerplate fault tolerance code has now been removed, so that it does not clutter the application's core business logic.

V. Case Studies

We claim that FT-REST provides reusability, programmability, and extensibility advantages when implementing fault tolerance functionality. To validate these claims, we have conducted two case studies, in which we applied FT-REST to realistic RESTful services, thereby rendering them fault tolerant.

The first case study validates the FT-REST reusability claim, while the second study demonstrates that FT-REST offers a programming model that is concise and extensible.

A. Reusing Fault Tolerance Strategies

In this case study, we analyzed the ability of FT-REST to reuse its strategy implementations across different RESTful applications. Reuse is useful only when the reused functionality retains its effectiveness. To that end, we demonstrate that the same FT-REST specification not only can be applied to different applications but also handles failure equally effectively for all the applications.

Our setup consisted of two synthetic services, with randomly injected faults. The first service was injected with network timeouts, while the second one with HTTP error codes. Thus, the fault characteristics of these services differ. Nevertheless, the fault tolerance of both of these services can be improved by applying the **Retry** strategy. When hand-coding the fault tolerance functionality, two separate implementations must be provided to account for the differences in fault detection. With FT-REST, we were able to apply the same FTDL specification to both services with only minimal changes and keep the applications running in the presence of the injected faults. Figure 5 shows these two FTDL versions, with the delta depicted in bold. Notice that the complexity of the services being rendered fault tolerant does not affect the required FTDL sophistication. Because well-designed RESTful services represent a single, isolated unit of functionality, this case study does not unfairly benefit our approach due to the simplicity of our synthesized test services.

B. Extending Fault Tolerance Strategies

To demonstrate the software engineering benefits of FT-REST, we show how our approach compares with the

```
<strategy>
  <service
    matchesUri="http://ft-rest.appspot.com/timeout"
    method="GET" />
  <conditions>
    <timeout>3000</timeout>
  </conditions>
  <sequential>
    <endpoint uri="http://ft-rest.appspot.com/timeout"
      method="GET" numRetries="5"
      backoffInterval="1000" backoffType="linear" />
  </sequential>
</strategy>
```

```
<strategy>
  <service
    matchesUri="http://ft-rest.appspot.com/status"
    method="GET" />
  <conditions>
    <status>503</status>
  </conditions>
  <sequential>
    <endpoint uri="http://ft-rest.appspot.com/status"
      method="GET" numRetries="5"
      backoffInterval="1000" backoffType="linear" />
  </sequential>
</strategy>
```

Figure 5. Reusing FT-REST specification files requires minimal programming effort.

	Traditional		FT-REST	
	Java	XML	Java	XML
Total LOC	199	0	32	39
Lines modified	12	0	0	14
Lines added	15	0	0	3

Figure 6. The lines of code in the two versions of fault tolerance.

hand-coded approach to implementing fault tolerance. Our subject application is an online banking service that exposes five equivalent REST endpoints, each with the following functionality: create a new account (PUT), add money to an account (POST), and retrieve an account balance (GET). To handle faults related to network volatility and service unavailability, each service invocation needs a unique fault tolerance strategy. One design option that can make this application fault tolerant is to apply the **Retry-Sequential** strategy to the create an account operation; the **Sequential-Retry** strategy to the add money operation; and the **Parallel-Retry** strategy to the retrieve an account balance operation.

To demonstrate the benefits of FT-REST, we developed two versions of this applications: (1) using hand-coded fault tolerance, and (2) using FT-REST introduced fault tolerance. It is desirable to reduce the amount of code that the programmer has to write. Thus, the fewer lines of code the programmer has to write, the higher are the perceived software engineering benefits.

Figure 6 shows the respective lines of code for the two versions. The “Total LOC” row represents the total lines of Java and XML code needed to create each version. Under the traditional approach, 199 total lines of Java code are needed. A majority of this code implements fault tolerance—parallel service invocation, exception handling, error code detection—–often obfuscating the core business logic. Using FT-REST, however, only 32 lines of Java and 39 lines of XML were required, thus drastically reducing the programming burden and removing the intermingling of fault tolerance and business logic code. Therefore, FT-REST effectively encapsulates fault tolerance as a separate and reusable FTDL specification file.

To show how straightforward it is to extend existing FT-REST strategies, we modified each aforementioned fault tolerance strategy and analyzed the impact on the source code. In particular, the strategies were changed as follows: (1) change the number of retries for each service invocation, (2) change the backoff interval, (3) change the backoff type, (4) add the HTTP status code of 503 as a failure condition, and (5) change the default timeout value. The last two rows in Figure 6 summarize the results. The traditional approach required modifying 12 non-consecutive lines of Java code, while FT-REST required no changes to the Java code and only 14 changes to XML, all of which were applied to consecutive sections of the FTDL file. In terms of new code, the traditional approach added 15 lines of Java code, while FT-REST added three lines of XML. Therefore, FT-REST can systematically extend fault tolerance with no changes to the core business logic, and minimal, straightforward changes to XML-based FTDL specifications.

VI. Discussion

FT-REST provides numerous software engineering benefits for the programmer faced with the challenges of constructing and maintaining resilient, fault tolerant RESTful applications. Compared to hand coded fault tolerance, FT-REST improves programmability, reusability, and extensibility. Additionally, FT-REST enhances the REST model with adverbs that describe service interactions.

A. Programmability

Separation of concerns is the holy grail of software engineering. FT-REST treats fault tolerance as a separate concern, enabling programmers to specify fault handling logic independently of the underlying business logic. The separation of the fault handling concern empowers the programmer to fully focus on implementing the core business logic, thus lessening the cognitive load and reducing the amount of code to be written and maintained.

As demonstrated, the characteristics of many fault tolerance strategies can be parameterized and expressed in a concise XML format. FTDL provides a major building block for FT-REST, as it equips the programmer with a consistent model for creating and deploying fault tolerance strategies. We argue that FTDL is easily learned by developers because it is based on XML, an industry standard. While some effort is required to extract the specific fault tolerance parameters for each service invocation, codifying these parameters in XML is much simpler than implementing the core logic needed to realize these strategies.

B. Reusability

FT-REST enables intra-application, inter-application, and cross-platform reusability. First, FTDL specifications are reusable across service invocations within the same application. With manual fault tolerance, when programmers add a new service invocation, they are responsible for manually inspecting each service invocation to ensure uniform fault handling. FT-REST enables them to simply add the invocation to the code, allowing the framework to ensure that the same strategy is used across all invocations. Hence, FT-REST reduces the amount of code that a developer needs to write and maintain by a factor of n, where n is the number of service invocations.

FT-REST also enables reuse across different applications. It is not uncommon for separate applications to use the same web service, albeit for different purposes. An application that locates restaurants in an area and an application that streams local tweets might both use a mapping web service. The fault characteristics of the mapping service will most likely be the same for both applications. And yet both applications might need vastly different mechanisms to handle faults. FT-REST enables the sharing of FTDL specification files, allowing fault tolerance to be written by experts and sparing the programmer from having to write the FTDL file.

Finally, FT-REST enables reusability across platforms. FTDL specifications are simply XML files, a universal data format with automated parsers in all major programming languages. While we have presented a Java variant of the FT-REST framework in this paper, the framework could just as easily have been written in C# or Python. Hand-written fault tolerance strategies are language-dependent, making porting an application to a new language tedious, as the fault handling code must be completely rewritten. However, FT-REST makes it possible to reuse FTDL files with any applications in any language; the FT-REST framework handles all the language-specific parsing and runtime interactions.

C. Extensibility

FT-REST provides systematic extensibility that enables the programmer to more robustly extend their fault tolerance strategies than manually adding fault handling code. Consider Figure 5 again. We wished to add the 503 HTTP status code as a failure condition for for a separate web service. While this could be trivially added in several lines of Java code if we chose to manually code fault tolerance, the task of manually changing several instances would unnecessarily burden the programmer. With FT-REST, the programmer only needs to add a single `<status>` tag and modify two other lines in the FTDL specification file to enable the needed fault tolerance for all service invocations.

D. Adding Adverbs to REST

REST is commonly described as applying verbs (HTTP methods) to nouns (URIs). The Web Application Description Language [17] and Web Services Description Language 2.0 [18] provide schemas for defining the endpoints, parameters, and responses that comprise a RESTful web service. These standards are used to describe each resource URI (noun) in greater detail and can be thought of as adjectives in the REST vocabulary. FT-REST qualifies the HTTP methods (verbs), delineating how a method should be called and reifying the requisite fault tolerance. Accordingly, FT-REST enhances the REST vocabulary with adverbs, which equip programmers with fine-grained control in defining and utilizing RESTful services.

VII. Related Work

The work in WS-DREAM [14] represents a unique approach to web service fault tolerance. The authors present a distributed fault tolerance framework that gathers QoS information of web services from geographically dispersed users. The QoS information is then input into a strategy selection algorithm to determine the optimal fault tolerance strategy to employ with a given web service. This design makes it possible to dynamically configure fault tolerance strategies. However, by aiming to completely automate the process, WS-DREAM disallows any involvement by the programmer in introducing application-specific fault handling code (e.g., backing up to an internal mirror, forcing a parallel invocation, etc.). FT-REST, however, provides programmers with the capability to specify application-specific fault tolerance strategies.

The Fault Tolerant Web Services Framework (FTWSF) [19] attempts to solve problems related to fault tolerance in web service invocations. The client-side C# library provides a set of classes and an XML-based language for defining retry and alternate URL semantics. For example, a programmer could specify that if a given web service does not respond, retry five consecutive times at intervals of five seconds. In addition, FTWSF provides a GUI that automatically produces the underlying XML configuration file. However, FTWSF does not provide semantics for trying multiple web services in parallel, and is targeted solely for SOAP web services. FT-REST builds off this work, extending it to include RESTful services and a more versatile set of fault tolerance strategies.

The FTWeb project [13] builds off the FT-CORBA project [20], bringing a fault tolerance infrastructure to web services. Mirroring the architecture of FT-CORBA, FTWeb replicates SOAP web service endpoints to provide fault tolerance, proxying all client web service requests by means of WS-Dispatchers, which maintain universal replica information to forward client requests to specific endpoints based on their health. If one replica goes down, the invocation is retried using a different endpoint and the WSDispatcher periodically queries the failed endpoint with `isAlive()` method calls until it is brought back online. This approach requires that programmers have control over the server as well, to respond to queries from WSDispatchers. By not relying on clients controlling the servers, FT-REST can add fault tolerance to independently administered services.

BPEL for REST [21] extends the Business Process Execution Language (WS-BPEL [22]) to include tags for sending and processing REST invocations. WS-BPEL allows programmers to declaratively structure business processes in an XML-like language, and have that specification translated into working code. Business processes traditionally used SOAP web services, but BPEL for REST introduces the `<get>`, `<post>`, `<put>`, and `<delete>` tags to enable business processes to use RESTful web services. Additionally, `<onGet>`, `<onPost>`, `<onPut>`, and `<onDelete>` tags represent the steps that servers should take upon receiving REST requests. Although BPEL for REST powerfully automates business processes, it lacks sufficient failure handling support. Although it can specify when a failure occurs, it cannot prescribe a specific handling strategy, thus burdening the programmer with the necessity to implement a fault handling mechanism.

The work in reference [23] presents a mechanism for presenting customizable and transparent durability in Service Oriented Architectures (SOAs). This work is similar to FT-REST in that the authors explore how best to treat service durability (i.e., fault tolerance) as a separate concern. They encapsulate durability policies within proxies that wrap web services to make their state durable. However, their approach relies on Java reflection, thus hindering its portability to other programming languages that may lack this facility. After the programmer implements these proxies, a separate durability mapping links proxies to a given service to provide fault tolerance. FT-REST extends this work by leveraging XML as a representation language for both durability policies and durability mappings. As a result, FT-REST features improved portability and reusability across platforms.

Salatge and Fabre [24] introduce fault tolerance connectors for unreliable web services. These connectors proxy *abstract web services*, defined as having numerous equivalent endpoints, and employ recovery strategies that exploit endpoint replication. The authors outline a model by which endpoints can be invoked passively or actively, depending on the desired fault tolerance strategy. Additionally they evaluate their work using several realistic web service applications. However, their approach works only for SOAP services. By contrast, FT-REST works with RESTful services and can also be extended to other distributed models.

VIII. Conclusion

As computing is moving from the desktop into the cloud, REST offers promising solutions to multiple challenges pertaining to constructing scalable and composable web services. Consistent interface definitions, scalability through replication, and the ubiquity of the HTTP infrastructure—all make REST a convenient architecture for a variety of applications. One implementation facet of RESTful applications that remains cumbersome is fault tolerance. RESTful applications must deal effectively with failures arising as a result of network volatility, service outages, and server errors. This need forces the programmer to code defensively, with the amount of fault tolerance code often exceeding that of core business logic. Furthermore, the resulting fault tolerance code is not amenable to encapsulation, being strewn throughout client applications. Thus, hand coded fault tolerance hinders programmability, reusability, and extensibility.

As an improvement over hand coded fault tolerance for RESTful applications, this paper has presented FT-REST, an architectural framework for enhancing RESTful applications with reusable and extensible fault tolerance. FT-REST effectively encapsulates fault tolerance strategies, systematically declaring and reifying them in a given application. To demonstrate the effectiveness of FT-REST, we showed how it can add reusable and extensible fault tolerance to realistic RESTful applications. As computing is becoming increasingly distributed, the issue of fault tolerance has come to the forefront of engineering the majority of computing applications. To that end, FT-REST presents innovative designs that can advance the state of the art in implementing fault tolerance across all distributed applications.

Acknowledgement

This research is supported by the National Science Foundation through the grant CCF-1116565.

References

[1] W3C, "Simple Object Access Protocol (SOAP) specification," Website, 2007, http://www.w3.org/TR/soap/.

[2] R. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, Citeseer, 2000.

[3] ProgrammableWeb.com, "Programmable web," website, 2011, http://programmableweb.com.

[4] Apache, "Apache HTTP components," Website, http://hc.apache.org/httpcomponents-client-ga/.

[5] Python Software Foundation, "httplib—HTTP protocol client," Website, http://docs.python.org/library/httplib.html.

[6] Oracle, "Jersey for Java," Website, 2011, http://jersey.java.net/.

[7] Ruby on Rails, “Ruby on Rails,” website, 2011, http://rubyonrails.org/.

[8] W. Brown, R. Malveau, H. McCormick III, and T. Mowbray, *AntiPatterns: Refactoring software, architectures, and projects in crisis*. Wiley, 1998.

[9] V. Dialani, S. Miles, L. Moreau, D. De Roure, and M. Luck, “Transparent fault tolerance for web services based architectures,” *Euro-Par 2002 Parallel Processing*, pp. 107–201, 2002.

[10] C. Fang, D. Liang, F. Lin, and C. Lin, “Fault tolerant web services,” *Journal of Systems Architecture*, vol. 53, no. 1, pp. 21–38, 2007.

[11] D. Liang, C. Fang, and C. Chen, “FT-SOAP: A fault-tolerant web service,” in *Tenth Asia-Pacific Software Engineering Conference, Chiang Mai, Thailand*, 2003.

[12] A. Liu, Q. Li, L. Huang, and M. Xiao, “FACTS: A framework for fault-tolerant composition of transactional web services,” *Services Computing, IEEE Transactions on*, vol. 3, no. 1, pp. 46–59, 2010.

[13] G. T. Santos, L. C. Lung, and C. Montez, “FTWeb: A fault tolerant infrastructure for web services,” *Enterprise Distributed Object Computing Conference, IEEE International*, vol. 0, pp. 95–105, 2005.

[14] Z. Zheng and M. Lyu, “Optimal fault tolerance strategy selection for web services,” *International Journal of Web Services Research*, vol. 7, no. 4, pp. 21–40, 2010.

[15] Twitter, “Streaming API Concepts,” Website, https://dev.twitter.com/docs/streaming-api/concepts.

[16] G. Dobson, “Using WS-BPEL to implement software fault tolerance for web services,” in *Software Engineering and Advanced Applications, 2006. SEAA’06. 32nd EUROMICRO Conference on*. IEEE, 2006, pp. 126–133.

[17] W3C, “Web application description language,” http://www.w3.org/Submission/wadl/.

[18] ——, “Web services description language,” http://www.w3.org/TR/wsdl20/.

[19] Manitra, “Fault tolerant web service framework,” Website, 2008, http://ftwsf.codeplex.com/.

[20] Object Management Group, “Fault tolerant CORBA specification,” Website, 2010, http://www.omg.org/spec/FT/1.0/.

[21] C. Pautasso, “BPEL for REST,” in *7th International Conference on Business Process Management (BPM08)*, Milan, Italy, September 2008, pp. 278–293. [Online]. Available: http://www.jopera.org/node/149

[22] OASIS, “Web services business process execution language version 2.0,” Website, 2007, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[23] X. Zhang, M. Hiltunen, K. Marzullo, and R. Schlichting, “Customizable service state durability for service oriented architectures,” in *Sixth European Dependable Computing Conference, 2006. EDCC’06*. IEEE, 2006, pp. 119–128.

[24] N. Salatge and J. Fabre, “Fault tolerance connectors for unreliable web services,” in *Dependable Systems and Networks, 2007. DSN’07. 37th Annual IEEE/IFIP International Conference on*. IEEE, 2007, pp. 51–60.