RAFT-REST - A Client-side Framework for Reliable, Adaptive and Fault-Tolerant RESTful Service Consumption

Josef Spillner, Anna Utlik, Thomas Springer, Alexander Schill

Technische Universität Dresden,
Faculty of Computer Science,
01062 Dresden, Germany
{josef.spillner,thomas.springer,alexander.schill}@tu-dresden.de,
anna.utlik@mailbox.tu-dresden.de

Abstract. The client/server paradigm in distributed systems leads to multi-stakeholder architectures with messages exchanged over connections between client applications and services. In practice, there are many hidden obstacles for client developers caused by unstable network connections, unavailable or faulty services or limited connectivity. Even if many frameworks and middleware solutions have already been suggested as corrective, the rapid development of clients to (almost) RESTful services remains challenging, especially when mobile devices and wireless telecommunications are involved. In this paper we introduce RAFT-REST, a conceptual framework aimed at engineers of clients to RESTful services. RAFT-REST reduces the effort to achieve reliable, adaptive and fault-tolerant service consumption. The framework is applied and validated with ReSup, a fully implemented flavour for Java clients which run on the desktop and on Android mobile devices. We show that by using the framework, message loss can be reduced significantly with tolerable delay, which contributes to a higher quality of experience.

1 Introduction

As the World Wide Web and the Internet of Services become more enriched with different kinds of information and functionality, private and commercial participants in these networks find new ways to expose their internal data, applications and even hardware resources. From the user's point of view, the interaction with such services is not restricted anymore to read-only access to information, but also includes utilisation and modification of data resources for creating complex applications and systems. Web services following the paradigm of Representational State Transfer, or simply RESTful web services, are attracting the attention of an increasing number of individual developers and service-driven companies as an easy and scalable way to provide an interface to different user groups. This has led to an enormous growth of the adoption of distributed RESTful applications. Many popular Internet-scale services like Facebook, Twitter, Dropbox, Flickr and Amazon S3 rely on the RESTful service paradigm.

This development is a result of major advantages RESTful services offer. Based on HTTP for resource access and service invocation in almost all practical realisations of the RESTful approach, overhead with auxiliary data otherwise used for encapsulating messages in envelopes and expressing operations [1] is reduced. Compared with SOAP-formatted and other procedural service protocols [10], RESTful services are easier to develop and use and fit more naturally the underlying stateful entity model like databases and hardware resources. Additionally due to the absence of the auxiliary data and the predictable nature of their requests they typically offer better performance. The latter property allows these services to be consumed with less restrictions on devices with limited computational power, such as mobile phones, tablets and embedded systems. Thus, with the simultaneous increase of radio communication and wireless network covered areas, RESTful services give their consumers a possibility to access and store valuable information in any place, at any time, and from the most comfortable terminal.

The other side of the coin is a high dependency on a stable and continuously available network connection, which raises a set of challenges for the development of RESTful service clients, especially when mobile devices and wireless telecommunications are involved. The addressed statelessness prevents services from keeping any information about clients on the service side, thus making the client responsible for maintaining the session integrity and handling its interruption due to network volatility and service failures. Mitigating this issue implies writing fault-handling code that is individual to every service. Additionally, a high dependency on external services makes mobile applications and other clients consuming RESTful services unusable during a network downtime, as no data which would be necessary for these applications can be received. It is left to the client developer if caching of data and its further reuse from a cache storage should be implemented in the application or not. However, the RESTful paradigm encourages caching and provides sufficient means to benefit from this technique.

As another drawback, the wild growth of services and client programming techniques for them has led to the lack of standardisation and formalised service description and engineering methodologies. Although the basic principles of REST are well known [4] and a lot of best practices and recommendations have been published in developer-oriented literature [11, 9, 14] to determine one good way of designing RESTful services, many implementations bypass some or even all those rules and hence require treatment of *almost-REST* anomalies.

The current approach to develop RESTful service clients is to use a service-specific framework offered by service providers. These frameworks provide a high-level API which might be the most convenient and appropriate solution if a client for a particular service should be implemented. As soon as a RESTful service client should consume services provided by different vendors service-specific frameworks are not appropriate. In addition, as we will discuss in the related work section, these frameworks only have a very limited support for fault-handling, disconnection support and configurability. Altenative approaches use low-level network programming based on HTTP or support access to RESTful services at message or service level. While these approaches are more general, flexibile, configurable and reusable, implementation effort is also significantly higher. Especially, advanced mechanisms for fault-handling, configuration, caching and adaptivity have to be implemented by the developer. Thus, development of RESTful service clients, especially for mobile devices, is currently a challenging, costly and time consuming task.

In this paper we therefore introduce RAFT-REST, a conceptual framework for reliable, adaptive and fault-tolerant access to RESTful services. Any implementation of it is targeted to the provision of service handling support on the client side in the context of unreliable network conditions combined with often sloppy almost-RESTful service interfaces. The conceptual framework proposes a novel way of accessing and consuming RESTful services from applications which cannot assume high-quality networks and service behaviour.

The contribution of the paper is threefold. Firstly, we describe the results of an analysis of 12 highly popular RESTful services and a set of fault-tolerant consumption frameworks with respect to support for the aformentioned problems. This analysis underpins the current state of service-specific frameworks as stated before. Secondly, we introduce the concepts of RAFT-REST as a conceptual, generic and portable client-side framework for rapid and cost efficient development of reliable, fault-tolerant and adaptive RESTful service clients. Thirdly, we describe ReSup, a concrete implementation of the RAFT-REST concepts for Java clients running on a desktop or Android device. We especially present and evaluation of fault-handling, configurability and performance for the use case of reliable mobile service consumption.

The remainder of the paper is organized as follows: In Section 2 we present and compare related client-side service integration approaches. Then, we introduce RAFT-REST as a concept and an architecture for reliable service consumption in Section 3. Through a reference implementation on Android, RAFT-REST is then evaluated for the use case of reliable mobile service consumption in Section 4. Finally, the performed work and its results are summarised and directions for future research towards more failure-aware service-oriented architectures are explained.

2 Related Work: SDKs and Fault-Tolerant Frameworks

Developers of clients for RESTful service interfaces have a choice in the level of assistance which coincides with the level of restriction to certain libraries, toolkits and providers. The following methodologies exist:

– Low-level network programming through APIs such as `socket` or, slightly more comfortable, on the HTTP level, such as `HttpUrlConnection`, `HttpClient` and `curl`. These libraries partially offer functions to overcome communication issues, but do not understand service response semantics.

- Message-level programming where each message is represented as a data structure. The data types are generated from message and interaction descriptions such as RPC IDL, Protocol Buffers, ggzcommgen or XML Schema. In this methodology, the message transport (e.g. HTTP or message buses) needs to be selected but not explicitly handled by the client developer.
- Service-level programming where messages descriptions are complemented by interfaces and endpoints, metadata about the provider and service context, as well as quantitative non-functional properties describing for instance the cost of a single invocation [8]. In this methodology, the most suitable message transport is selected automatically.
- Provider-level programming by using a provider-specific software development kit (SDK). This methodology trades the ability to switch flexibly between providers for a simplified programming model, in particular avoiding the need to configure endpoints within the application. Sometimes, the SDKs even offer graphical elements aimed at instant service consumption by humans.

None of these practical methodologies are well-suited for imperfect network conditions and generic service faults. Researchers have found several API and SDK shortcomings and proposed improvements. We will summarise the findings and extend them with our own observations. Additionally, we will highlight some proposed service consumption concepts which focus on reliability and fault-tolerance, and explain why they don't fully match the needs of developers of applications in imperfect networks.

2.1 Conventional consumption frameworks

RESTful services and programmable web APIs are expected to follow certain established guidelines and best practices. Among them are (1) well-designed URIs as resource identifiers, (2) well-projected request semantics on top of the method semantics which the transport channel provides, (3) appropriate status codes, (4) conscious use of metadata along with each request and response, and (5) correspondence between content types and well-formed messages.

Vendors of services typically offer SDKs for a variety of programming languages, frameworks and platforms. These SDKs are almost always restricted to a single service endpoint. Furthermore, they mirror all design weaknesses which violate the guidelines for service design.

As a complement to previous API and SDK analysis work [11, 14, 2], we have performed an up-to-date analysis of 12 currently highly popular RESTful services which confirms the view that more robust and reliable integration and consumption techniques are needed. The analyis encompasses social interaction (Shuffler.fm and Facebook), data storage (Dropbox and Amazon S3), images (Flickr and Daisy), audio (SoundCloud) and video (Vimeo), open data (Open 311) and security and management of resources (CloudPassage and Sun Cloud). These services differ not only in their domains, but also in technical characteristics including the data model, message body formats, CRUD operations (Create/Read/Update/Delete), HTTP headers and HTTP response codes. Further differences exist in their adaptivity, security, internationality and documentation.

Guideline violations among this set of APIs are plentyful, hence we can only briefly mention some notable examples. In the Facebook Graph API, POST is used for both creations and updates, while PUT is not used. In Flickr, the semantic difference between POST and GET is expressed with a GET parameter. Amazon S3 transmits only stock response codes in the headers and delivers meaningful error symbols in the body, and Flickr omits the codes altogether.

Table 1 summarises the SDK characteristics for services whose SDK offers at least some fault tolerance or caching.

Table 1. Result of client libraries evaluation

	Facebook	Dropbox	Amazon S3	Flickr	Sound-Cloud
Representation of resource data model as API types	+	+	+	+	−
Wrapping of received exceptions into verbose error messages	+	+	+	−	+
Request retry techniques	−	−	+	−	−
Handling of connection or read timeouts	−	+	+	-	−
Data integrity checking	−	−	+	−	−
Caching of successful responses	+	+	−	−	−

2.2 Fault-tolerant consumption frameworks

Researchers have found existing SDKs to be essential for a quick adoption of new services among application developers, and yet insufficient due to their restriction to one service, mirroring of service design weaknesses, and assumption of perfect networks [7, 3]. In this section, four proposals with partially existing framework or code implementations will be presented and compared: iTX, FTWeb, DR-OSGi and FT-REST.

The issue of maintaining a stable Internet connection in wireless networks is a well known problem of mobile devices. Modern distributed applications, typically developed on web services, often rely on mobile components, which heavily depend on different types of wireless connection (WIFI, GPRS, UMTS, Wi-Max, etc.). Apart from instability of applications, for mobile devices individually such network problems can cause performance and monetary costs. Many research efforts have been started in this area in order to achieve resilience of such application against network volatility.

The authors of the mobile recovery concept [15] describe the problem of mobile client state recovery after the network disconnection and reconnection. The goal of this work is to reduce the costs of recovery if a user was involved in a long-lasting network transaction which was interrupted. The notion of Internet transaction (iTX), as used in this work, has been taken from the familiar concept of database transactions and describes the user interaction with one or more network resources for achieving one or more objectives. The proposed solution involves logging of user state for each step of iTX. As initial steps in transaction can branch into several parallel and independent subtransactions, it is necessary to track in which subtransaction the user is at any given moment, and evaluate which actions should and can be recovered. Therefore, the proposed solution allows to reuse the current state from the log and does not repeat the steps of transactions again, thus saving network traffic. However, the algorithm of the given solution requires continuous computation and updating of an action graph, as well as persistent storage for user states. This is why mobile devices, due to limited battery, memory and processing power, are a bad choice for placement of this solution.

The authors of the FTWeb project [13] and the Fault Tolerant Web Service Framework [12] both provide a fault tolerant layer for unreliable web services. The model proposed in FT-Web provides a software layer that acts as proxy between client requests and service responses. This proxy ensures transparent fault handling for client by usage of active replication. The given approach addresses requirements of high service availability and reliability for distributed systems. In contrast, the Fault Tolerant Web Service Framework project proposes customisable fault-tolerance connectors between client and service. Each such connector is a software component which captures web service interactions and partially performs built-in fault tolerance actions. It encapsulates pre-processing and post-processing of requests, and recovery actions which can be parameterised by user. Connectors reside on a third-party infrastructure between service providers and consumers. An infrastructure assumption on which the given approach relies is redundant services. This is used by recovery strategies which implement passive and active replication of request between equivalent services. Both of these approaches are targeted at SOAP web services, and involve third party components between clients and services.

The bundle-oriented DR-OSGi [6] proposes a systematic handling of network outages in distributed system in a consistent and reusable way by implementing fault tolerant strategies as reusable components. This solution is targeting service-oriented applications implemented on top of the OSGi platform. Hardening strategies for network volatility resilience, such as caching, hoarding, replication, etc., are provided as OSGi bundles, and could be added to existing applications transparently. Although the presented benchmarks are showing improved results with DR-OSGi when network becomes unavailable in exchange for a small performance overhead, this solution affords needless system resources consumption. This circumstance prevents it from being widely used on mobile devices. Also, the underlying OSGi framework restricts the usage of DR-OSGi on platforms where OSGi is not supported, or must be pre-installed manually, for instance on Android.

The work in FT-REST [3] presents an approach to fault handling in client RESTful applications. This proposal consists of a domain-specific Fault Tolerance Description Language (FTDL) and client-side library. FTDL is used to specify fault tolerance policies for RESTful application which are compiled afterwards by FT-REST framework into platform specific code. This code module can be added to the business logic of applications and acts as a fault-tolerant layer between the application and a RESTful service. The authors claim that this concepts brings benefits in programmability – by separating the fault tolerance concern from the underlying logic, the programmer can fully focus on implementing the core of application; reusability – by reusing the XML-structured FTDL specification of a service between different applications and across platforms; and extensibility – by robust extension of FTDL fault-tolerant strategies instead of writing fault-tolerant code. FT-REST encapsulates the following fault tolerance strategies: retry (reattempting a service endpoint), sequential (iteration through the set of equivalent endpoints), parallel (simultaneous invocation of equivalent endpoints), and composite (combinations of the aforementioned). However, client-side caching and other disconnected operations techniques are not considered in this work.

Many scientific works are focused on adding resilience to distributed network applications, but only few of them consider the client as a point of network and service fault handling. Among the introduced works, FT-REST can be considered as highly related, as it provides fault handling support for the client and is aimed at consuming RESTful services. However, no previous approaches have been found which provide developers with an out-of-the-box solution that helps to implement RESTful mobile application with necessary fault tolerance.

3 Service Consumption Concept

In this section, the design choices and concepts behind a framework for reliable, adaptive and fault-tolerant RESTful service consumption (RAFT-REST) will be presented. This includes a detailed description of the logical and the derived structural architecture, a service consumption workflow and a consideration of component interactions.

3.1 Logical Architecture

RAFT-REST is a conceptual framework which ensures a high level of reliability, resilience, adaptivity and fault tolerance for service consumption subject to network and service failures. It is intended to achieve service-oriented architectures of higher quality, in particular for mobile clients and embedded cyber-physical systems connected to often brittle service interfaces in the cloud.

The high-level logical architecture of RAFT-REST is shown in Fig. 1. It consists of an API Mapping Component, a Request Executor, a Network Manager as well as a Cache and a Request Queue. The purpose of the *API Mapping Component* is to transform requests and messages into a format understood by the service interface. The *Network Manager* monitors the network connections maintained by the client device and notifies the other componentens of the RAFT-REST framework about changes of connectivity. The heart of the RAFT-REST logical architecture is the *Request Executor*. As an active component it manages the Cache and Request Queue and handles incoming request from clients as well as response messages from services. Access to the network is managed based on the connectivity information provided by the Network Manager. The *Request Queue* fulfills the requirement of asynchrony by enqueuing all outgoing request received from a client. The *Cache* is responsible for temporarily storage of requested resources in the persistent memory of the client device. Thus, it is a key component to handle short-time network failures and longer disconnection phases.

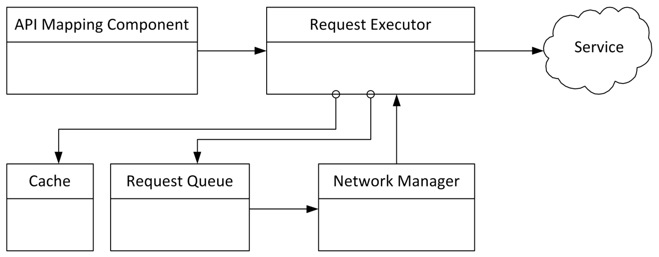


Fig. 1. RAFT-REST Logical Architecture

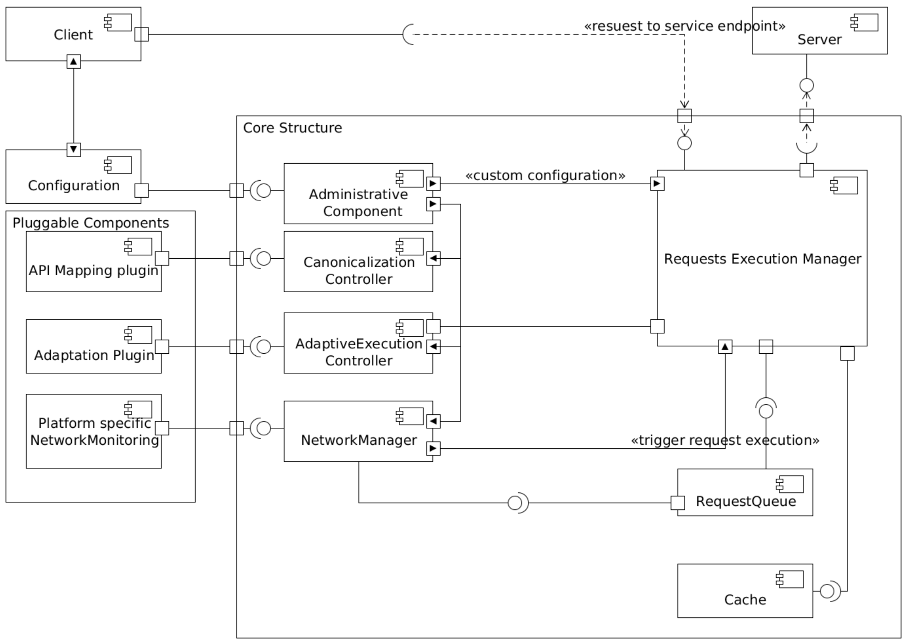
As a particular design choice, the framework should be as generic as possible, while allowing for almost-REST service-specific workarounds. Therefore, messages can be adapted by the API mapper when instructed to do so by the application. A typical use case is to substitute an error code for a success indication in cases of occurred problems.

3.2 Structural Architecture

The structural architecture of RAFT-REST, which is derived from the logical one, is shown in Fig. 2. It describes the framework from the perspective of a developer.

Client and Service. The Service component represents a RESTful service, which provides a RESTful service API. The Client component represents the Application client which intents to consume the RESTful service.

The Core Structure of the framework is build around the *RequestQueue* for request messages and the *Cache* for resource representations. The **Request Execution Manager** acts as mediator with fault tolerance and adaptation functions. The fault tolerant part checks network and message errors, including response error codes, makes decisions about whether such errors can be mitigated, and applies fault handling techniques. The adaptive part is mostly specific to certain services. It adapts resources to the device context apart from general adaptations, e.g. to work around wrongly modelled response codes. The Request Execution Manager provides an interface, via which clients can pass requests and receive responses. During the instantiation a client can configure some of the network and performance related parameters using the **Configuration** component.



Client
«resuest to service endpoint»
Server
Core Structure
Configuration
Administrative Component
«custom configuration»
Pluggable Components
API Mapping plugin
Canonicalization Controller
Requests Execution Manager
Adaptation Plugin
AdaptiveExecution Controller
Platform specific NetworkMonitoring
NetworkManager
«trigger request execution»
RequestQueue
Cache

Fig. 2. RAFT-REST Structural Architecture

The **Network Manager** is aware of network conditions, and supply this information to other components, which rely on it (i.e. Requests Queue and RequestExecution Controller). As it is possible to have more than one type of network connection on mobile devices, and retrieving information about connection state is associated with inquires to hardware, Network Manager must rely on **Network Monitoring** components, which are platform specific. Thus, the framework can be extended with custom Network Monitoring components at this point.

The **Canonicalization component** is present in the architecture to supply the framework with canonical entities of the requests and responses. The most widespread service descriptions now exist as plain text, which can be understood by human. In such situation RESTful service endpoints, corresponding HTTP methods and responses should be transformed manually by the developer. Pluggable components can be added to achieve automatic transformation.

Finally, the **Administrative component** collects statistics and manages the level of information that is printed into log messages.

3.3 Service Consumption Workflow

The intended workflow for RAFT-RESTful service consumption is represented in Fig. 3 to illustrate the behavior of the RAFT-REST framework. The complete workflow is organized into three phases.

In *Phase 1* the client initiates the interaction with a service by handing a service request over to the framework. As a first step this request is transformed into a canonical representation which allows the framework to process different types of service interfaces in a common way. As a second step, the cache is checked. If the reply to this request is found in the cache, e.g. the result of an idempotent request which is always cacheable, it is returned immediately to the client.

Otherwise, *Phase 2* is entered and the request is placed into a request queue where it stays until an active network link to the service is available. For mitigating the absence of a connection, a random delay time is added before messages can be forwarded using a re-established network connection to avoid request overhead after service ior network failures.

In *Phase 3* the resource is requested from the service and eventually error responses could be received. In this phase the framework is responsible for evaluating the cause of an error and deciding about appropriate error handling. For instance a retry of a message can be triggered. Parameters for error handling such as the maximum number of retries and the back off time between retries are configurable. Finally the response is handled (e.g. response data is cached) and forwarded to the initiating client.

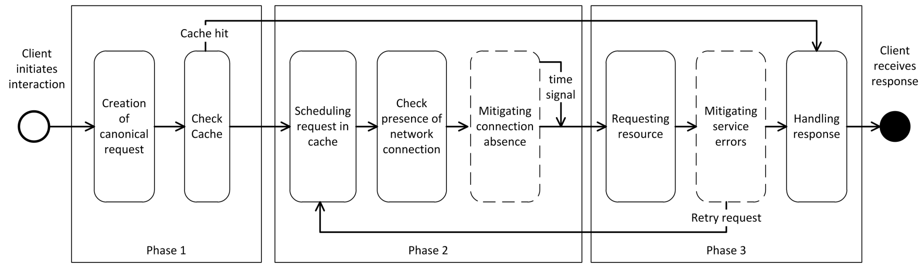

Client initiates interaction
Creation of canonical request
Cache hit
Check Cache
Phase 1
Scheduling request in cache
Check presence of network connection
Mitigating connection absence
time signal
Phase 2
Requesting resource
Mitigating service errors
Handling response
Retry request
Phase 3
Client receives response

Fig. 3. Workflow of RESTful service consumption according to the RAFT-REST concept

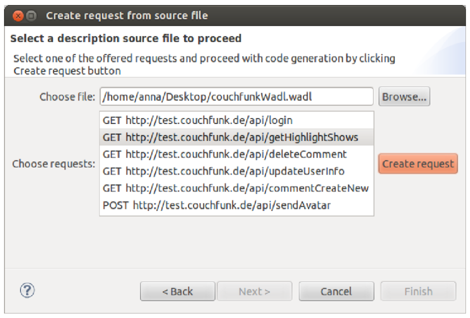
Table 2 offers a comparative summary of the characteristics of RAFT-REST and FT-REST. While RAFT-REST lacks a structured description of faults, for which a domain-specific language is yet to be defined, it offers additional functionality especially through caching and error mitigation. Equivalent endpoint invocation is a feature already found in general adaptive service proxies and therefore can be combined with RAFT-REST while keeping the framework lean.

Table 2. Comparison of characteristics: FT-REST vs RAFT-REST

Characteristics	**FT-REST**	**RAFT-REST**
Separation of concerns - separating of fault-tolerant layer from application logic	yes	yes
Fault condition description	for each endpoint, in XML document	for each endpoint (optionally hierarchical), in generated classes
Handling strategies configuration	for each endpoint	for all application
Timeout handling	yes	yes
HTTP response error codes handling	yes	yes
Mitigation of errors, embedded in the message	no	yes
Service binding code	no	yes
Fault handling techniques		
Retrying of request	yes	yes
Equivalent endpoints invocation	yes	no
Cache	no	yes
Asynchronous request invocation (due to network conditions)	no	yes

4 Validation: The ReSup Framework

ReSup (RESTful Support), a Java framework for the development of service-bound mobile and desktop applications, turns the RAFT-REST concepts into practice for service clients implemented with the Java language and corresponding networking libraries. ReSup consists of a proxy library located between the application and the network stack, similar to existing HTTP client libraries, and an Eclipse wizard to generate service-specific message classes by mapping API specifications to code. A screenshot of the wizard in Fig. 4 highlights the possiblity to import WADL (Web Application Description Language) service descriptions to reduce the manual class modelling effort [8]. Both the wizard and the library need to be present as JAR files (`org.tud.resup.apimapper_1.0.0.2013MMDD.jar`, `Resup_1.0.0.2013MMDD.jar`) in the plugins folder of Eclipse. The library is subsequently copied into the resulting project archive and used at runtime.



Create request from source file

Select a description source file to proceed

Select one of the offered requests and proceed with code generation by clicking Create request button

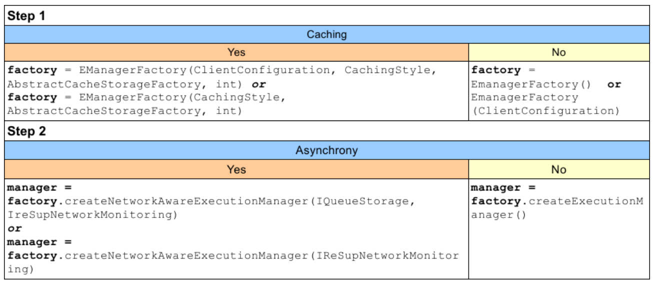
Choose file: /home/anna/Desktop/couchfunkWadl.wadl

Choose requests:

GET http://test.couchfunk.de/api/login
GET http://test.couchfunk.de/api/getHighlightShows
GET http://test.couchfunk.de/api/deleteComment
GET http://test.couchfunk.de/api/updateUserInfo
GET http://test.couchfunk.de/api/commentCreateNew
POST http://test.couchfunk.de/api/sendAvatar

Fig. 4. Eclipse wizard to generate API mapping classes for use with the ReSup library

The ReSup library delivers and caches message objects which may be created manually in the application code or as instantiations of the generated classes. Each message object represents a service resource or a request targeting one. The way ReSup is used depends mainly on the application requirements. Fig. 5 demonstrates the possible configuration directive combinations.



Step 1	
Caching	
Yes	No
factory = EManagerFactory(ClientConfiguration, CachingStyle, AbstractCacheStorageFactory, int) ***or*** **factory** = EManagerFactory(CachingStyle, AbstractCacheStorageFactory, int)	**factory** = EmanagerFactory() **or** EmanagerFactory (ClientConfiguration)
Step 2	
Asynchrony	
Yes	No
manager = **factory**.createNetworkAwareExecutionManager(IQueueStorage, IreSupNetworkMonitoring) ***or*** **manager** = **factory**.createNetworkAwareExecutionManager(IReSupNetworkMonitoring)	**manager** = **factory**.createExecutionManager()

Fig. 5. Flexible configuration of the ReSup objects depending on the desired application behaviour

The integration of ReSup is particularly easy by its HttpClient interface which mimicks the API of the widely used Apache `HttpClient` library. Hence, developers can switch to ReSup by just substituting a Java import statement and afterwards gradually turning on and testing its RAFT features. ReSup can also run as a transparent stand-alone proxy so that applications do not have to be modified except for a system-wide forced HTTP proxy configuration. Proxies, gateways and networked intermediaries are common architectural elements for RESTful service consumption [5]. However, on mobile systems such as Android, this requires system modification access and is therefore often not a viable solution.

In order to demonstrate the capabilities of ReSup, and therefore acknowledge the RAFT-REST concepts, we have performed a number of measurements and experiments in an evaluation study which includes existing services with RESTful interfaces in both simulated and real imperfect networks. Fig. 6 shows the setup of the experiment. Off-the-shelf networking tools such as Burp, an intercepting proxy, Trickle, a bandwidth variator, and Netem, a packet loss generator, are used to simulate reproducible low-quality connections. Trickle allows scaling the bandwidth within the constraints imposed by the hardware from a zero-throughput connection to the typical speeds of mobile networks, WLANs and LANs. Netem varies typical WAN parameters such as delay, loss, duplication and re-ordering (shuffling).

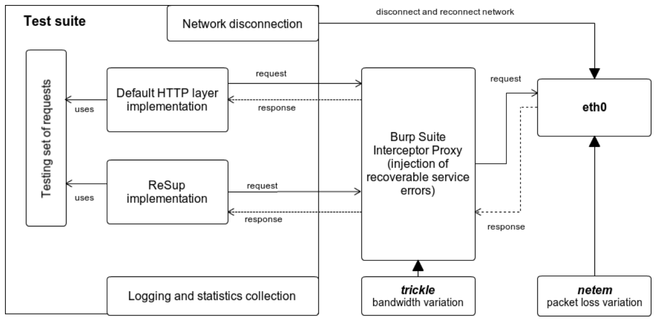


Fig. 6. Test suite for the experiments involving RESTful services and clients using the ReSup library

The test environment consisted of the test suite running on an Intel Core i5 machine with 4 times 2.4 GHz CPUs, 3 GB DDR2 RAM, the Ubuntu 12.04 operating system and Java 1.6.

The results are shown in Fig. 7 and 8, respectively. The first diagram shows the distribution of error responses still successfully received from one of the test runs, in this case a mobile connection with 50 kbps. On the whole range of possible network losses, ReSup retrieves a higher number of error reponses compared to a pure HttpClient connection. The increase is between 35% and 92%, but still recovers less than half of the lost packets when the loss rate exceeds 80%. The second diagram measures the overall response times over a bivariate range from a perfect connection (100 Mbps with 0% loss) to a nearly unusable one (50 kbps with 80% loss). Due to the caching, ReSup is much faster for all good-enough connections and considerably slower than HttpClient for all worse ones due to the retries. Yet, the retries contribute to the higher success rate as shown in 7, therefore even the highest run-time overhead of 55,4% for a 500 kbps connection with 60% loss will eventually improve the user experience in practice.

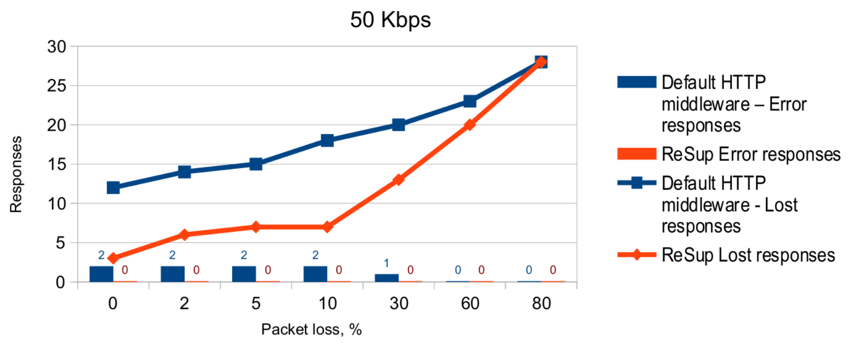


Fig. 7. Error response numbers evaluation

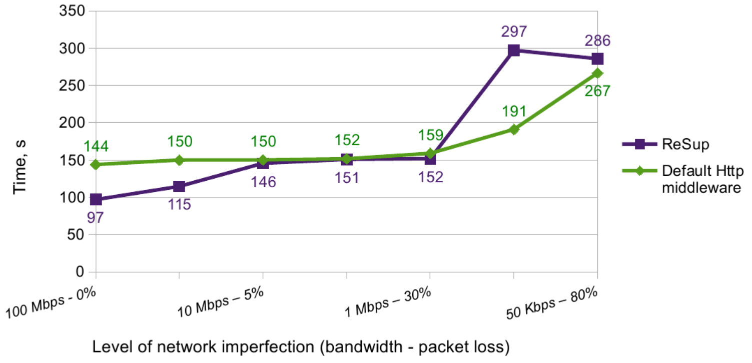


Fig. 8. Response times evaluation

5 Conclusion

We have motivated and discussed RAFT-REST, a client-side RESTful service integration concept which addresses reliability, safety and quality concerns. Compared to previous integration concepts, the combination of conventional fault-tolerance schemes, graceful error handling and caching achieves a great user experience for distributed applications connected by imperfect networks. The ReSup framework targets mobile application developers through an API-compatible HTTP client library and an Eclipse code engineering plugin with the aim to increase real-world application robustness through RAFT-REST. It is made available as open source toolkit for use and further improvements[1].

Subsequent research questions focus on the global view, i.e. the server-side and total load behaviour when using any of the offered RAFT-REST mechanisms, as well as on extended validation with real-time statistics collection and visualisation for immediate feedback about the waiting state of applications to the user.

[1] ReSup website and source code: `http://serviceplatform.org/wiki/ReSup`

Acknowledgements

This work has received funding under project number 080949277 by means of the European Regional Development Fund (ERDF), the European Social Fund (ESF) and the German Free State of Saxony.

References

1. Aihkisalo, T., Paaso, T.: Latencies of Service Invocation and Processing of the REST and SOAP Web Service Interfaces. In: Eighth IEEE World Congress on Services (SERVICES). pp. 100–107 (June 2012), Hawaii, USA
2. Belqasmi, F., Glitho, R.H., Fu, C.: RESTful Web Services for Service Provisioning in Next-Generation Networks: A Survey. IEEE Communications Magazine 49(12), 66–73 (December 2011)
3. Edstrom, J., Tilevich, E.: Reusable and Extensible Fault Tolerance for RESTful Applications. In: 11th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). pp. 737–744 (June 2012), doi: 10.1109/TrustCom.2012.244
4. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine (2000)
5. Kelly, M., Hausenblas, M.: Using HTTP Link: Header for Gateway Cache Invalidation. In: Proceedings of the First International Workshop on RESTful Design (WS-REST). pp. 23–26 (April 2010), Raleigh, North Carolina, USA
6. Kwon, Y.W., Tilevich, E., Apiwattanapong, T.: DR-OSGi: Hardening Distributed Components with Network Volatility Resiliency. In: Proceedings of the ACM/IFIP/USENIX 10th International Conference on Middleware. Lecture Notes in Computer Science (LNCS), vol. 5896, pp. 373–392 (December 2009), Urbana Champaign, Illinois, USA
7. Leitner, P., Rosenberg, F., Dustdar, S.: Daios – Efficient Dynamic Web Service Invocation. Internet Computing 13(3), 72–80 (May/June 2009)
8. Maleshkova, M., Pedrinaci, C., Domingue, J.: Supporting the creation of semantic RESTful service descriptions. In: 8th International Semantic Web Conference (ISWC) (October 2009), Washington D.C., USA
9. Masse, M.: REST API Design Rulebook. O'Reilly (2011)
10. Mulligan, G., Gracanin, D.: A comparison of SOAP and REST implementations of a service based interaction independence middleware framework. In: Proceedings of the Winter Simulation Conference (WSC). pp. 1423–1432 (December 2009), Austin, Texas, USA
11. Mulloy, B.: Web API Design: Crafting Interfaces that Developers Love. e-Book (March 2012)
12. Salatge, N., Fabre, J.C.: Fault Tolerance Connectors for Unreliable Web Services. In: 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 51–60 (June 2007), Edinburgh, UK
13. Santos, G.T., Lung, L.C., Montez, C.: FTWeb: A Fault Tolerant Infrastructure for Web Services. In: Proceedings of the Ninth IEEE International EDOC Enterprise Computing Conference. pp. 95–105 (September 2005), Enschede, The Netherlands
14. Tilkov, S.: REST Anti-Patterns. InfoQ Article (July 2008)
15. VanderMeer, D., Datta, A., Dutta, K., Ramamritham, K., Navathe, S.B.: Mobile User Recovery in the Context of Internet Transactions. IEEE Transactions on Mobile Computing 2(2), 132–146 (April–June 2003)