EdgStr: Automating Client-Cloud to Client-Edge-Cloud Transformation

Kijin An  
Software Engineering Team  
Samsung Research  
Seoul, South Korea  
kijin.an@gmail.com

Eli Tilevich  
Software Innovations Lab  
Dept. of Computer Science, Virginia Tech  
Blacksburg, VA, United States  
tilevich@cs.vt.edu

Abstract—To harness the potential of edge resources, two-tier client-cloud applications require transformation into three-tier client-edge-cloud applications. Such transformations are hard for programmers to perform correctly by hand. Many cloud services maintain a runtime state that needs to be replicated at the edge. Once replicated, this state must then be synchronized efficiently and correctly. To facilitate the transition to edge computing, we present a framework that automatically transforms client-cloud applications to client-edge-cloud versions. Our framework, EdgStr, automatically replicates cloud-based services at the edge. EdgStr synchronizes the replicated service state by relying on a third-party Conflict-Free Replicated Data Type (CRDT). It generates code that connects service state changes to CRDT update operations, thus ensuring that the state changes at each replica eventually converge to the same replicated state. As an evaluation, we applied EdgStr to transform representative distributed mobile apps for deployment in dissimilar network and device setups. EdgStr correctly replicates cloud services (targeting the important domain of Node.js), deploying the resulting replicas on an ad-hoc edge cluster, hosted by Raspberry PI devices. As long as eventual consistency is congruent with the functionality of a cloud service, EdgStr can automatically replicate this service and deploy the replicas at the edge, thus offering the performance benefits of edge-based execution, without the high costs of manual program transformation.

I. INTRODUCTION

With its superior and elastic computing resources, cloud-based execution has long provided performance and scalability benefits to remote clients. With the emergence of edge computing, transforming a two-tier (cloud/client) application into its three-tier (cloud/edge/client) counterpart can become an effective means of enhancing scalability and performance [1], [2]. When such transformation takes place, one must take into account the inherent differences in resource availability between cloud and edge environments. While cloud environments are resource-abundant, edge environments are typically resource-scarce. In addition, edge devices are often connected via unstable and limited networks [3].

To take advantage of edge resources, distributed applications can be developed from scratch. However, a more common scenario is when developers need to introduce edge processing to existing cloud applications as a performance optimization [4]. To integrate edge-based processing, an existing cloud-based application is modified, so it retains its functionality while its architecture is transformed from two-tier (client-cloud) into three-tier (client-edge-cloud). To transform such an app correctly by hand, so its performance would improve as intended, is non-trivial. Developers first have to understand the app’s distributed runtime behavior to identify those cloud-based functionalities that can benefit from edge-based processing. Then, developers have to determine whether these functionalities maintain their state and how to keep it consistent once replicated. Finally, developers have to manually modify distributed communication protocols, commonly implemented using special frameworks with complex APIs.

To facilitate this process, this paper introduces EdgStr, a novel approach that automatically replicates cloud services at the edge, thereby transforming two-tier applications into their three-tier variants. EdgStr operates by first instrumenting live HTTP traffic between the client and the cloud to determine the available services for replication. It then identifies the subset of the server code to replicate via profiling and constraint solving. Finally, it replicates live the relevant server state, thus transforming the original client-cloud application to integrate edge replicas, whose states are kept eventually consistent with each other and that of the cloud server. To provide the required eventual consistency for the newly replicated state, EdgStr takes advantage of a third-party Conflict-Free Replicated Data Type (CRDT) [5]. It generates code that connects the service’s state changes to the corresponding CRDT update operations. The CRDT’s built-in eventual consistency ensures that the connected replicated state’s changes are reliably and efficiently propagated across the participating edge replicas.

In summary, EdgStr brings the performance advantages of edge processing to pre-existing two-tier applications, eliminating the need for costly manual program analysis and transformation. By describing EdgStr’s design, implementation, and evaluation, this paper makes the following contributions:

1) A novel approach for seamlessly integrating edge resources into pre-existing client-cloud applications. The approach involves analyzing the calling patterns of cloud services and identifying specific functionalities within these services that can be replicated at the edge. Doing so enables leveraging the differences between the available WAN and LAN capacities to enhance performance.
2) A reference implementation of our approach, EdgStr, which supports the popular Node.js cloud services.

3) An evaluation of EdgStr’s automatic transformation of third-party client-cloud applications into their client-edge-cloud counterparts, showcasing its effectiveness in:
   a) minimizing synchronization requirements for replicated service states;
   b) reducing response latency in replicated services;
   c) managing fluctuations in client request traffic and minimizing device energy consumption.

The rest of this paper is organized as follows. Section II presents a motivating example and describes our approach; Section III discusses the design and implementation of EdgStr; Section IV presents our evaluation and its findings; Section V compares our approach with the related state of the art; and Section VI presents concluding remarks.

II. MOTIVATION AND APPROACH

We start by presenting a motivating example and then introduce EdgStr and its redistribution model.

A. Motivating Example

Figure 1 depicts the dataflow of a third-party distributed mobile app, firebase-objdet-node, which is distributed across a mobile client and a cloud server. Specifically, the client captures images and sends them to the server for processing. The server analyzes the received images, returning the analysis results back to the client. To that end, the server performs computationally intensive operations that localize and identify the constituent objects in the received images, guided by the pre-trained model of a deep learning framework. The server then transfers the results to the client, which uses them to draw the boundaries of and descriptions around the identified objects in the captured images. Assume that the app is deployed for a mission-critical task, such as security monitoring, so it is essential to achieve the requisite levels of response latency.

Modern smartphones typically come with 8-16 megapixel cameras. An image captured with such cameras would typically take between 1 and 20 Mbytes. As images are being transferred to the cloud-based image processing service, the network’s bandwidth and latency determine the resulting performance. In addition, providers of cloud services can host them in different geographic regions, not necessarily those co-located with the client. The service’s actual geographic location can drastically affect the round-trip time (RTT) metric. To demonstrate this insight, we installed our example app’s remote service on the cloud infrastructures, located on the same continent and on the nearest neighboring continent\(^1\). The RTT across different continents is an order of magnitude larger than within the same continent. Because the app is mission-critical, experiencing such a slowdown in performance would cause the app to fail in its mission.

To work around the network bottleneck conditions described above, one can take advantage of edge computing resources, provided by nearby network-connected computing devices. Such devices can be accessed in a single hop within the shared local network but would offer less processing power than their cloud-based counterparts. In other words, some image processing can be offloaded to nearby edge devices for processing, so the app would adaptively take advantage of edge computing resources for certain combinations of processing loads and RTTs.

B. EdgStr Framework

The EdgStr framework performs a behavior-preserving automated program transformation. EdgStr attaches to a running client-cloud application and captures its live HTTP traffic between the client and the invoked cloud-based services. Based on the captured traffic, EdgStr then analyzes and transforms the original two-tier application to a semantically equivalent three-tier application. The transformed application continues delivering the original functionality, but with improved latency and throughput. The transformation dynamically generates and instantiates a Remote Proxy: the edge node becomes the cloud server’s filtering and processing proxy, preserving the original service interfaces. In brief, certain functionality of a cloud-based service becomes replicated at edge nodes, co-located with clients. The service states are then synchronized between the cloud and the edge replicas in the background.

One could simply duplicate the entire server-side functionality at the edge nodes, synchronizing all of the replicated service states. Because such a naïve approach would incur unacceptably high synchronization costs, the amount of functionality to transfer to and execute by the edge replicas must be carefully selected. Also, failure handling logic may not be easily transferable from the cloud to the edge, requiring managing complex states. To allow for easy replication at the edge, the edge replicas handle failures by forwarding the failed service invocations to the master service copy in the cloud. This scheme reduces the overhead and complexity of not only handling failure but also of synchronizing states.

\(^1\)We used Heroku as our cloud provider
C. Redistribution with Remote Proxy

The Remote Proxy pattern [6] describes an architectural style in which a client communicates with a placeholder for some remote functionality. In distributed computing, proxies are used widely to adapt or enhance various remote services. A Proxy object serves as a surrogate for a real remote service RealSubject and exposes the same access interface Subject to its clients Client (Figure 2). The power of the Proxy pattern lies in its ability to control the interaction between the client and the remote service, without having to modify them. In terms of providing the actual functionality, a proxy can either forward all client requests to a remote service or provide some of the services itself. In distributed systems, proxies have been used to implement functionalities ranging from caching to authentication. In our approach, the proxy object is responsible for determining whether a client request is to be forwarded to a remote server or is to be serviced in place. The requests serviced in place take advantage of edge processing.

D. Applicability and Limitations

Obviously, EdgStr cannot replicate all cloud services, so we clarify its applicability and limitations. The EdgStr’s design relies on a RESTful HTTP protocol that follows request/response patterns. If a service is invoked via HTTPS, EdgStr would need to apply its packet-level sniffer and analyzer after the server has decrypted the received commands from HTTPS to plain HTTP, an arrangement that may need assistance from system administrators.

To keep replicas consistent, EdgStr generates code that connects the state changes at each replica to a third-party CRDT’s update operations. The CRDT keeps the resulting replicated state eventually consistent. CRDTs provide what is known as strong eventual consistency [5]. Strong eventual consistency might not be congruent with the requirements of all cloud services, thus limiting the applicability of our approach. Nevertheless, we argue that EdgStr is widely suitable for the types of services that process client-collected sensor data. Such services are CPU-bound, transforming sensor data collections into computed summaries, persisted for future referencing. The functionality that such services are expected to deliver allows their replicas to tolerate being in temporarily inconsistent states, with the CRDT-provided strong eventual consistency meeting their functional and non-functional requirements. While EdgStr can replicate JavaScript objects, disk files, and relational databases, it is inapplicable to non-SQL databases, such as MongoDB. Finally, EdgStr relies on the conventions of the Node.js framework, currently supported by Google Cloud, Amazon Web Services, and Microsoft Azure, with adopters including LinkedIn, Netflix, Uber, and PayPal.

III. DESIGN AND IMPLEMENTATION

We implement our approach, as the EdgStr framework, whose general flow and main components appear in Figure 3. We describe these components in turn.

A. Analyzing HTTP Traffic

A modern Web application’s predominant interaction paradigm is the RESTful architecture [7]. Cloud-based components expose REST APIs, comprising a set of HTTP methods applied to different URLs, invoked by client components. Our approach automatically generates the Subject S part of the Proxy pattern based on dynamic analysis. Assume RealSubject S comprises N externally invokable RESTful services s1, ..., sN. To determine the Subject S access interface, our approach starts from instrumenting all HTTP traffic between Client and Real Subject in order to decode the parameters of the client’s HTTP requests p1, ..., pN and the return results of the server’s HTTP responses r1, ..., rN, with the assumption of responses being non-empty.

\[
S = [s_1(p_1) \cdots s_N(p_n)] = [r_1 \cdots r_n] \quad (1)
\]

Next, the actual subset of Real Subject is identified. For this subset of remote functionality, each remote service si’s executed code is identified and profiled. First, the entry/exit points of the identified server code of si become the boundary for the Extract Function refactoring, which places that code into an individually invocable functional unit. By identifying all the executed statements, the refactoring then infers possible execution states of si for Real Subject. These states are necessary to determine what and where to synchronize in the subsequent proxied remote execution.

B. Capturing Relevant Server States/Code

EdgStr replicates a subset of the cloud server’s state on edge servers. To determine which subset to replicate, it analyzes the runtime traces of the execution of cloud-based services. Consider a typical life-cycle of a cloud service: (1) the server initializes itself to the init state state\text{init} and (2) the server receives an HTTP request from the client and unmarshals the passed parameters p_i (unmarshaling, i.e., \text{img} in Figure 1), (3) the server passes the unmarshalled parameters to the invoked service, which starts executing s_i, and (4) upon completing its execution, the service marshals the response to return it over HTTP to the client r_i (marshaling, i.e., \text{ret} in Figure 1). For brevity, we refer to the above steps (2)(3)(4) as the \text{i}th execution or exec\_i. To infer the subset of the cloud server’s state to replicate at edge proxies for the subject \text{i}th REST API s_i, EdgStr not only extracts the code executed in step (2), but also captures the subset of the cloud server’s state required to initialize the edge replicas.
What complicates this procedure is that some cloud-based services are stateful, and as such change their state with every execution. Hence, it becomes problematic to detect their relevant subsets of the state to replicate. Even if such stateful services could be restarted anew for every execution, only some of their states would be restored, as they also often persist data in a database.

Therefore, EdgStr isolates state changes when analyzing the dynamic execution traces of server-based services. This state isolation ensures that the server’s init state \( \text{state}_{\text{init}} \) and the service’s execution results remain fixed. After executing \( \text{init} \), EdgStr checkpoints the resulting state, so it can be restored before executing the service again:

\[
\text{init}, \text{exec}_i, \text{exec}_{i+1}, \ldots
\]

\[
\text{init}, \text{save} \, \text{“init”}, \text{exec}_i, \text{restore} \, \text{“init”}, \text{exec}_{i+1}, \text{restore} \, \text{“init”}, \ldots
\]

C. Isolating and Replicating States

EdgStr identifies and replicates several different units of cloud-based services that typically include “a database”, “files”, and “program variables” [8], [9].

Database Tables: EdgStr monitors dynamic traces of a Node.js cloud-based service with jalangi [10], a dynamic JavaScript analysis framework. To identify database-related statements, EdgStr instruments all function invocations whose argument values are SQL commands. To that end, EdgStr modifies the INVOKEFUNCTION(LOC, F, ARGS, VAL) callback API of Jalangi to be able to examine the arguments parameter, ARGS. Then, it adds shadow executions into the identified SQL invocations LOC to trace the changes to the database state. First, EdgStr appends shadow execution of the original SQL command with a SQL command to snapshot the entire database. Next, it adds transactional executions START TRANSACTION/ROLLBACK against the original SQL command, to keep the database tables unchanged.

Files: In cloud-based services, files can be accessed both locally and remotely. To identify file accesses, EdgStr instruments all invocations whose arguments are file URLs. It then duplicates the identified files by copying or downloading.

Global variables: EdgStr adds get/set function instrumentation after the declarations of global variables to implement their save and restore operations. After the server has been initialized, EdgStr deeply copies all global variables and saves their states. The restore operation passes the saved states to each variable’s set function.

D. Determining Whether Eventual Consistency Is Acceptable

In distributed systems, the question of which consistency model is suitable for replicated data is highly application-specific. It would be unrealistic to be able to answer this question outside of the programmer’s purview. Hence, we rely on the programmer to determine if EdgStr can be applied successfully. To that end, EdgStr isolates the state that would be replicated, as described in Section III-C. EdgStr then presents the isolated state information (as specific source code statements) to the programmer, who then decides whether the presented state can tolerate being kept consistent via an eventual consistency model, as that provided by CRDTs (the Consult Developer step in Figure 3). If eventual consistency would satisfy the service’s requirements, the programmer then goes ahead and applies EdgStr that replicates the given cloud service at the edge. If the programmer misidentifies the suitability of eventual consistency, then EdgStr-generated replication might not preserve the user-expected functionality of the original cloud service. In the presence of a solid testing procedure, this mistake should be straightforward to identify and correct by manually implementing a stronger consistency protocol for the replicated service.

E. Identifying Server Code to Replicate

For Subject, \( S \) and \( S \)’s \( n \) remote services \( s_1, \ldots, s_n \), EdgStr identifies the subset of Real Subject’s server code, in terms of specific code statements, that need to be replicated at the edge replicas. EdgStr identifies the entry/exit points of service executions. Since only application logic is extracted, without any extraneous functionalities such as fault handling, EdgStr instruments the entire server-side code, but uses only those instrumentation results that are generated by the successful service executions. The resulting entry and exit points of \( s_i \) become the boundary for the Extract Function refactoring, which places the extracted code into a standalone function. By identifying all the executed statements, EdgStr then summarizes the statements and the execution states of \( s_1, \ldots, s_N \) for Real Subject. To determine the entry/exit points, EdgStr...
identifies the \textit{marshaling} and \textit{unmarshaling} functions that process parameters and return values. Marshaling (the exit point) converts program values to a data format suitable for network transmission; unmarshaling (the entry point) reverses the process by converting the transmitted data format to regular program values [11].

Consider the remote service "/predict" (Figure 1). Although this service's application logic is not explicitly delineated at the function boundary, \textit{EdgStr} can still extract it by analyzing the execution of \texttt{exec 1}, discovering the unmarshaling parameter \texttt{p1} and the marshaling result \texttt{r1}. Specifically, to identify these entry/exit points, \textit{EdgStr} normalizes the entire server code by introducing temporary variables. For example, \textit{EdgStr} normalizes \texttt{var tv1 = new Uint8Array(b);} and \texttt{res.send(tv2);} as shown in Figure 4 (left). \textit{EdgStr} examines the app's execution logs for all read and write operations that involve the \texttt{p1} and \texttt{r1} values. This examination reveals that the variable \texttt{tv1} holds \texttt{p1}, and \texttt{tv2} holds \texttt{r1}. To differentiate between the primitive type values related to the analyzed service and those used by unrelated functionalities, \textit{EdgStr} fuzzes the HTTP messages, so the parameter \texttt{p1} becomes \texttt{p1[1], \ldots, p1[i]} and the modified messages are tracked by means of a fuzzing dictionary. \textit{EdgStr} identifies the entry/exit statements as those that read/write the fuzzed values. Between the entry/exit points, \textit{EdgStr} extracts all dependent statements and their states into a new, independently invokeable function (Figure 4).

\textit{EdgStr} conducts its dependence analysis by means of declarative logic programming. It represents JavaScript statements and how they relate to each other as logical facts and predicates, respectively. \textit{EdgStr} also generates special facts to represent the statements that unmarshal a parameter and marshal a return value. The RW-LOG($s_1,v_{i1},p_1$) fact expresses that the variable $v_{i1}$ in the statement $s_1$ reads or writes $p_1$. \textit{EdgStr} encodes the RW-LOG-FUZZED fact that describes the read/write behaviors of the $i^{th}$ fuzzed version of RW-LOG. The STM-T-UNMAR rules infer the entry point that unmarshals $p_1$, in which the read/write events occur in the same position. Similar to STM-T-UNMAR, the predicate STM-T-MAR is used to infer the return value $r_1$.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4.png}
\caption{Normalization and Extracted Function}
\end{figure}

\textit{EdgStr} extends the STM-T-DEP rule to be able to identify where function $f$ is declared. Notice that the ACTUAL fact expresses the invocation of function $f$.

\begin{equation}
\text{STM-T-DEP}(s_1,s_2) \leftarrow \text{STM-T-DEP}(s_1,s_2) \land \text{STM-T-DEP}(s_2,s_3).
\end{equation}

\textit{EdgStr} uses a control-flow graph between the statements by using POST-DOM($s_1,s_2$), which expresses that statement $s_1$ post-dominates $s_2$. Expressed in terms of STM-T-DEP, POST-DOM is:

\begin{equation}
\text{STM-T-DEP}(s_1,s_2) \leftarrow \text{POST-DOM}(s_1,s_2)
\end{equation}

The following Algorithm 1 describes how \textit{EdgStr} extracts the relevant application logic from the server program:

\begin{algorithm}[h]
\caption{Overall \textit{EdgStr}'s Dependency Analysis}
\begin{algorithmic}
\State $S \leftarrow \text{parseHTTPa}(H)$ \Comment Subject from HTTP records $H$
\State $\text{state}^{\text{init}} \leftarrow \{\}$ \Comment initial state, extracted functions, and dependent statements for $S$
\State $\text{forall } p_0, r_1 \in S \text{ do}$
\State \Comment Query unmarshaling JS statement and its variable
\State \Comment Query marshaling JS statement and its variable
\State \Comment Query all dependent statements in entry/exit points
\State \Comment Extract Function Refactoring for $s_i$
\State \Comment Merge all Execution results
\State \Comment Replicate (copy) the necessary init state of $S$ based on Jalangi framework
\EndForAll
\State $\text{state}^{\text{init}} \leftarrow \text{repStates}(St_{all})$
\end{algorithmic}
\end{algorithm}

\textit{EdgStr} identifies the entry/exit points by querying the statements that satisfy the constraints, defined by the STM-
UNMAR/STMT-MAR rules. Then, it queries for all dependent statements within the identified boundary. EdgStr applies the Extract Function refactoring to the remote service $s_i$; it copies the code fragment $St_{s_i}$ to create function $ftn_{s_i}$. Then it adapts $St_{unnmar}/St_{mar}$ to pass parameters and return a result at $v_{unnmar}/v_{mar}$ (see Figure 4). Finally, EdgStr merges the dependencies of all identified statements in services $s_1, ..., s_N$, replicating only the necessary cloud-based init state by means of the dynamic analysis described in Section III-C.

F. Synchronizing States

Figure 5 presents the original two-tier distribution model (left) and its three-tier counterpart (right), generated by EdgStr. In the generated three-tier model, each client communicates with an edge node in its local edge environment. Thus, each proxy edge node and the cloud node run replicas of the same service. This description, of course, is a simplification, as they are not exact replicas but rather share some state. Each node can modify this shared state concurrently, so these modifications need to be synchronized.

![Figure 5: Distribution Models](image)

Because the edge nodes and the cloud server communicate via a WAN, whose connectivity may not remain constant, our synchronization follows a weak consistency model. Distributed systems with weak consistency models allow replicated states to diverge temporarily. The divergent replicated states then become eventually merged [16]. To provide the required weak consistency, EdgStr generates code that connects the service’s state changes to the corresponding CRDT [5] update operations, as detailed in Section III-G. A CRDT’s predefined data structure achieves state convergence with mathematically sound update strategies. EdgStr’s relaxed consistency semantics allows the replicated state to be synchronized in a background process without interfering with the provisioning of main functionalities [17].

G. Program Transformation

1) Transforming Cloud-based Service: To efficiently synchronize the states across the edge replicas and the cloud-based server, EdgStr takes advantage of CRDTs [18]. Specifically, EdgStr wraps the replicated components ‘database’, ‘files’, and ‘global variables’ into ‘CRDT-Table’, ‘CRDT-Files’, and ‘CRDT-JSON’, respectively. CRDTs provide all the required machinery to keep the replicas eventually consistent. To keep track of changes and resolve conflicts, these CRDT-structures provide the API calls of initialize, getChanges, and applyChanges. To synchronize states between the cloud service and its edge replicas, EdgStr initializes 1) both the master and the replicas with the same snapshot of the cloud-based service and then 2) keeps transmitting/applying changes between the cloud and its edge replicas. The cloud server periodically sends its state changes to each edge node (via cloud_state messages in Figure 5-(b)), while each edge node notifies its state changes to the cloud server (via edge_state messages in Figure 5-(b)). Specifically, EdgStr identifies all the affected code statements in each replicated component, rewriting them by means of CRDT templates\(^2\). The changes between the cloud and its edge replicas are efficiently exchanged via the bidirectional socket.io API.

2) Generating Edge Replicas: Given a list of cloud-based server functions and a state snapshot, EdgStr then generates replicas to be deployed on edge nodes. To generate readable code that can be tweaked by hand, EdgStr uses the handlebars template framework\(^3\). To keep the states between the edge replicas and the cloud service eventually consistent, each edge node initializes its CRDT data structure with a passed state snapshot. Once generated and deployed, a replica starts proxying for the cloud-based service when processing client requests. It executes a service and responds to the client, while asynchronously transmitting the state changes by means of its CRDT data structure to the cloud-based service, which serves as the cloud-based master copy. As a consequence, the CRDTs at the replicated edge nodes unconditionally accept all changes received from the cloud-based CRDTs. The CRDT mechanism of EdgStr guarantees that all replicas would eventually arrive to the same state, a property that is tolerant to temporal state divergence or merging delay between the edge nodes. However, with relaxed convergence, concurrent updates could still result in dissimilar final states.

IV. EVALUATION

Our evaluation seeks answers to the following questions:

- **RQ1. Correctness** Does EdgStr preserve the functionality of original client-cloud apps when transforming them to their client-edge-cloud versions?
- **RQ2. Performance** How does EdgStr affect the throughput, latency, and energy consumption of apps? What impact does EdgStr’s clustering have on latency and energy consumption when it comes to edge devices?
- **RQ3. Efficiency** How accurately does EdgStr’s dynamic analysis identify the portion of the replicated service state that must be synchronized? How does EdgStr’s proxying compare with existing distributed proxying techniques in terms of performance overhead?

A. Subject Applications

We evaluated our approach with 7 open-source distributed applications and their 42 remote services. To identify these subjects, we searched GitHub for repositories for distributed applications, in which both the client and the server communicate via HTTP. To that end, we searched the results

\(^2\)https://github.com/automerger/automerger

\(^3\)https://github.com/handlebars-lang/handlebars.js
after each service execution, we reset the original and the replicated versions of subject services to their *init state* via EdgStr-generated restore operations. Executing the original regression tests against all subject services did not reveal any discrepancies between the original services and their replicas produced via EdgStr (42/42).

### C. Performance

We evaluated how network conditions and device processing power impact application performance (RQ2). Here, we present the results of evaluating performance in terms of throughput, latency, and consumed energy in turn.

**Network Setup:** Our system utilizes two network links 1) *an edge network:* this local area network has a strong signal strength of -55dBm or better. 2) *a cloud network:* this wide area network (WAN) is configurable with offset parameters for delay and bandwidth to exhibit different network characteristics using a system-level network emulator. For evaluating EdgStr in a *limited cloud network* setup, we set the bandwidth: [100, 1000] Kbps and the latency: [100, 1000] ms, similar to the environment of our motivating example scenario (Section II-A).

**Cloud/Edge Nodes Setup:** The processor speed of the cloud server far exceeds that of the edge nodes, invoking cloud services that transmit large volumes of data over limited network bandwidths. We therefore used DELL-OPTIPLEX5050 as our cloud infrastructure as well as RPI-3 (Raspberry Pi-3) and RPI-4 (Raspberry Pi-4) devices as our edge nodes (Figure 6-(a)).

Recall that the primary motivation for replicating cloud-based services at the edge is to reduce latency and increase throughput under varying network conditions. Hence, in our measurements, we established a baseline for the throughput rates as those that are typically reached under *good network conditions* (i.e., representative of typical edge network bandwidth). To ensure that our subjects exhibit normal performance characteristics, we applied a linear regression analysis to the cloud and the edge throughput rates. Because the slopes in

---

(b) Benchmarking Throughput and Latency

![Figure 6: Cloud/Edge Nodes and Mobile Device Setup](image)

---

Based on combinations of keywords for popular server and client HTTP middleware frameworks for Node.js, curated by the community. For server-side keywords, we used ‘Express’, ‘Koa’, etc., while for client-side keywords we used ‘Ajax’, ‘fetch’, ‘reactJS’, ‘Angular’, etc. The selected subjects use these frameworks to implement the communication logic of their respective client and server parts. Several subjects also make use of server-side databases and the TensorFlow framework. Table II describes our subject applications and details the experimental results; it shows the evaluated remote services with their HTTP verbs, alongside their network traffic and latency measurements.

**B. Correctness of EdgStr’s Replication**

We first checked whether the replicated services continue to deliver the same functionality as their original cloud-based versions (RQ1). Specifically, we compared if the replicated and original versions of our test subjects returned the same results for a given set of parameters. Given \((p_1, \ldots, p_n)\) sent to the original service \(OS\) and the replicated service \(RS\), check if \(R_{OS} = R_{RS}\). To that end, we utilized the regression test cases that come with the original apps, consisting of clients invoking HTTP requests against remote services. For this experiment,

---

### Table II: Subject Services and Their Refactored Services

<table>
<thead>
<tr>
<th>Subject Apps</th>
<th>Services (HTTP verbs)</th>
<th>WAN Traffice/Overhead(KB)</th>
<th>Latency(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fobojet</td>
<td>/predict(fGET)</td>
<td>22780, 2280</td>
<td>0, 0</td>
</tr>
<tr>
<td>mist-rest</td>
<td>/predict-d(fGET)</td>
<td>22740, 0.784</td>
<td>0, 0</td>
</tr>
</tbody>
</table>

**Cortex-A72**

- **Cloud Infra** (Desktop): 177-7700 (3.6GHzX8)
- **Edge Node** (RPI-3): Cortex-A53 (1.4GHzX4)
- **Edge Node** (RPI-4): Cortex-A72 (1.5GHzX4)
- **Mobile Dev** (Android): Snapdragon

---

https://github.com/tylertreat/comcast
Figure 6-(b) are far smaller than $y = x$, one can assume that all original subjects are well-optimized for a typical deployment environment: a powerful server connected via a reliable network. In addition, we applied a linear regression analysis between the edge nodes running RPI-3 and RPI-4. According to a processor benchmark [19], the CPU power of RPI-4 exceeds that of RPI-3 by a factor of 1.8, which is similar to our measurement value of 1.71 (0.075/0.044) as shown in Figure 6-(b).

1) Throughput: For each experimental subject, we compared the throughput (numbers of requests per second) achieved by the original client-cloud versions and their client-edge-cloud variants. We deployed original subjects in dissimilar network environments, in terms of the speed of their cloud links. To that end, we configured their bandwidths from 0.1 to 5 MBytes/s for our limited networks (Section IV-C). In a fast WAN, client-cloud always achieved higher throughput than their client-edge-cloud variants. As the WAN’s speed decreased, so did the client-cloud’s throughput, reaching a threshold at which the client-edge-cloud variants started achieving higher throughput. The performance advantage of edge-based execution manifests itself most prominently in subjects with relatively heavy upload/download data traffic or low computational loads (Figure 7-(a), (c), (d), and (f)). We demonstrate this by quantifying the transmitted data.

a) Data Deluge Index: To obtain deeper insights into how the amount of transmitted data affects the performance of cloud-based execution, we introduced Data Deluge index, a new metric, defined as $I_{deluge} = \Delta Net/\Delta Tput$. The $Tput$ variable represents normalized throughput for Figure 7, which scales the app’s original throughput to the range between 0 and 1. $I_{deluge}$ is defined as the network resources $\Delta Net$ needed to increase $Tput$. $I_{deluge}$’s increases for the original cloud service ended up being proportional to the amount of transmitted data, whereas the volumes of transmitted data over WAN did not affect $EdgStr$’s throughput (Figure 7-(g)).

2) Latency: Similarly to throughput, the network speed heavily affects the latency of mobile clients invoking cloud-based services. First, we established the baseline by profiling the respective invocation latency under favorable network conditions as $L_o$—the original cloud-based service, and $L_e$—its edge-based counterpart (see Table II for detailed results). Notice that $L_o$ was always smaller than $L_e$. As WAN’s conditions degraded, executing edge-replicated services became faster than executing their original cloud-based versions.

3) Energy Consumption: We also evaluated our approach’s impact on reducing the energy consumption of client devices. Because, nowadays, client devices are typically mobile and battery-powered, energy efficiency has become an important system design consideration. Notice that while a mobile client is waiting to receive the results from a cloud-based service, the mobile device typically switches into a low-power mode in the idle state to consume less energy. Nevertheless, the longer it takes to execute a cloud-based service, the more energy the client device will end up consuming, despite being in the low power mode [20].

We used Treep Profiler [21] to measure the consumed energy of an Android device running the Snapdragon chipset. We executed each subject 200 times and collected the profiled results for battery power (Watt) over the limited cloud network, as described in Section IV-C. As compared to the original client-cloud versions of our subjects, their client-edge-cloud versions consistently decreased their energy consumption by factors that range from 6.65J to 7.98J (Figure 8).

D. Evaluating the Scalability and Elasticity of Edge-Based Processing

To be able to distribute the incoming service requests from clients equally across the available edge replicas, we constructed a small cluster of edge devices, each hosting a service replica. To manage the cluster, we introduced a load balancer, a system module providing the following two capabilities: (1) optimize the cluster’s processing load by directing client request traffic to the edge nodes with the fewest active connections [22]; (2) estimate the expected volume of traffic by monitoring the number of active connections between edge

Figure 7: Cloud Network Speed versus Throughput Performance and Deluge Index (Section IV-C)

Figure 8: Comparison for Consumed Energy of a Mobile Device (Poor Network Setup)
nodes and their clients. We then configured the balancer to monitor the volume of client requests, so the running edge service replicas could be dynamically created and shut down in response to the changes in service utilization. Rather than completely shutting down an edge device, we instead put it into the low-power mode, so it could be brought back to the running mode without incurring unnecessary delays. Our load-balancing heuristic assumed that all client requests had similar processing requirements. As the number of active connections changed, the load balancer adjusted the number of active edge nodes accordingly.

To evaluate our approach’s effectiveness, we experimented with different execution scenarios. These scenarios emulated several distinct volumes of client requests to a remote service with various workloads that involved different read and modify functions. Finally, we evaluated how much energy could be saved by the edge devices to extend their battery life [23], [24]. To that end, we compared our approach to a naïve edge processing setup, measuring how much time the Rasberry PIs spent in the power-off battery mode.

Increasing the number of participating edge nodes lowered the network communication latency, especially in the presence of heavy network traffic (Figure 9-left). However, as the network traffic dissipated, engaging additional edge nodes in the execution neither decreased latency nor increased throughput. Rather, the attendant extra resource consumption can drain the battery power of the participating edge devices faster, as they must remain active to be ready to process both client requests and the cloud server’s synchronization requests. As is always the case with replicated services, the volume of synchronization requests increased proportionally with the number of replicas, which in this deployment were hosted by edge nodes.

Our evaluation setup comprised four edge replicas, each hosted by a Raspberry PI device, connected directly to the edge router by a wireless network (It consisted of 2 RPI-3s and 2 RPI-4s). In this performance benchmark, we measured the observed latency and energy consumption per client requests per second (RPS). Specifically, we varied the RPS from 10 to 300 in increments of 50. As expected, for higher RPS (from 200 and up), increasing the number of active edge replicas ended up decreasing the overall latency. In contrast, for lower RPS (between 10 and 200), the number of active edge replicas had no visible bearing on the observed overall latency.

The ability to power down the unused edge replicas led to noticeable energy savings. To measure the consumed energy, we attached a digital power meter to the edge devices. In response to the decrease in the volume of client requests, the number of active replicas gradually changed from 4 to 1, thus reducing the volume of consumed energy by as much as 12.96%, with the overall latency increasing only slightly (Figure 9-right).

E. Effectiveness of EdgStr

To answer RQ3, we compared the effectiveness of EdgStr’s synchronization and proxying strategy to those of other representative approaches used in distributed systems.

1) Effectiveness of EdgStr’s Synchronization: While network conditions can affect the performance of invoking remote services, stateful services behaved dissimilarly between invocations across all execution environments. (See Original WAN traffic $W_{AN_o}$, as shown in Table II). The need to synchronize the state changes affected the original WAN traffic patterns. When a replicated stateful service was executed at an edge node, the changed state was then asynchronously transmitted via WAN to the cloud server. We report both the minimum and the maximum amounts of EdgStr’s WAN ($W_{AN_e}$ in Table II). As shown in Figure 10-(a), we compared the amount of network traffic WAN, generated by the original version during a regular remote execution vs. the maximum WAN traffic resulting from EdgStr’s synchronization (Kbytes/request). For the majority of our subjects, EdgStr’s transformation reduced the amount of WAN traffic generated by a single service invocation. This reduction is due to our subject apps being data-intensive, with client-generated data transferred to the remote server for processing. Most of the cross-Instruction Set Architecture (cross-ISA) offloadings systems [25], [26], [27] synchronize the entire program state stored in the working memory ($S_{app}$ in Table II). As compared to those systems, EdgStr minimizes the amount of synchronization traffic over WAN by replicating only the modifiable parts of the replicated service state. Figure 10-(a) shows that, as compared to the cross-ISA systems, EdgStr reduced the synchronization overhead by orders of magnitude.

2) Comparing EdgStr to Caching/Batching Proxy Techniques: Caching: Proxy Caching [28], [29] can be particularly beneficial for read-mostly services. If the replicated service data is simply cached, it can then be accessed with low latency. However, caching may be inapplicable for replicating certain remote services. In the presence of state changes, the cached service data can become stale fast [30]. Besides, for example, some subjects would be unable to cache their data as $\{client\_param$ and $cloud\_result\}$. Because these subject services take images (i.e., hand-written digits or camera pictures) or text as input, their data has unique characteristics that would be impossible to duplicate. We discovered that only ‘Bookworm’ and ‘med-chem-rules’ could be cached.


Batching: A batching proxy aggregates multiple client requests to forward them as a single message containing the aggregated data to the server, which also returns the results in bulk [31], [32]. Hence, because batching reduces the number of WAN transmissions, it is most effective in high-bandwidth networks. However, if the volume of transmitted data saturates the available bandwidth, batching becomes ineffective. We aggregated the multiple remote executions by using distribution patterns—Data Transfer Object (DTO) and Remote Façade [33]—to quantify how these proxying strategies performed in comparison to EdgStr in terms of the total latency it took to invoke subject services. The measurement results appear in Figure 10-(b). Each benchmark was executed over the limited cloud network setups, as described in Section IV-C, with the average latency of batching between 2 and 10 executions reported.

As it turned out, all evaluated proxy strategies ended up reducing the response latency, as compared to the baseline cloud-based executions. Batching decreased latency by the smallest amount, due to our setup’s network bandwidth being smaller than the amount of the batched transfers’ aggregated data. Caching achieved the smallest latency for the min, Q1, and median benchmark. However, for the max and Q3 benchmarks, caching ended up increasing the latency as compared to the unproxied cloud-based baseline. Because many services cannot be cached at all, this proxying strategy’s applicability is rather low. EdgStr exhibited the lowest latency for most benchmarks with a few exceptions, in which caching showed lower latency (i.e., min, Q1, and median).

F. Threats to Validity

Internal Threats: We assumed that the replicated cloud services could handle failures effectively. In EdgStr’s replication, edge replicas relied on the cloud to handle failures. That is, a replica only detected failures but handled them by redirecting the failed service request to the cloud, assuming that the original cloud service would either succeed in executing the request or handle failures effectively. If this assumption about the failure handling capabilities of the original cloud services did not hold, our failure handling strategy of the edge replicas would be inadequate.

External Threats: Our experimental setups used Raspberry Pis as edge devices, which might not be as computationally powerful. However, what this design choice means is that our experimental setups did not unfairly advantage our approach. If we instead used more powerful edge devices, without changing any other aspects of our evaluation, our approach’s performance improvements would strictly increase.

V. RELATED WORK

EdgStr is related to several techniques and approaches introduced in prior works. We briefly describe them in turn.

Distributing Applications by Transforming Code: Similarly to our approach, program analysis and transformation have been applied to split single-tier applications into distributed multi-tiers. The ZQ compiler [34], Code Phage [35], and RT-Trust [36] transform centralized applications into their distributed counterparts to enhance security and privacy. The cross-ISA offloading frameworks [25], [26], [27] generate distributed applications whose different parts exchange their memory address mappings. Tango [26] generates replicas of a mobile app that bidirectionally initiates the communication between the client and the server, as driven by execution time conditions. Buffet [27] compiles a complex program into a subset of variants to run cross-ISA systems. After that, Buffet verifies the correctness and integration of the resulting cross-ISA distributed applications. CodeCarbonReply [37] and µScalpel [38] remove irrelevant functionality and integrate the extracted parts of one program with another program. However, the majority of these related works require that programmers manually annotate the programs to identify and transform the replicated states. Additionally, their program transformations can be applied only to centralized applications. In contrast, EdgStr requires no programmer annotations and takes distributed, client-cloud applications as input.

State Replication in Distributed Systems: Several frameworks replicate data of mobile clients for performance and fault tolerance. Legion [39] replicates data across clients by means of CRDTs. By interacting peer-to-peer, Legion’s clients reduce communication latency, remaining resilient in the presence of server disconnections. EdgStr shares the design objectives for replicating data over limited networks, but its novelty lies in automating the replication and consistency functionalities, eliminating the need to modify client or server code by hand. Fuzzing techniques have been applied to capture possible execution states of a cloud service. Rester [40], [41] analyzes APIs and execution logs to increase execution coverage or to identify security vulnerabilities in cloud-based services in the presence of dependencies across REST APIs. However, these approaches cannot replicate both program code and state, as required for replicating cloud services at the edge.

Optimization Techniques: Several systems offer batching optimization. Bouquet [42] detects and bundles together repeated requests. APE [43] defers distributed calls until the mobile device switches into the network activation state. However, the underlying network can cause batching transmissions to hurt performance, increasing latency and memory consumption. A
Stream Processing Engine (SPE) efficiently processes high rates of sensing data by distributing stream processing across workers [44]. To improve their performance within a job, workers can be controlled in a queue with a back-pressure algorithm [45], maximizing data locality [46], or accelerating network transfer [47]. These techniques can also benefit our approach if the underlying OS and architecture support them.

VI. CONCLUSIONS

We presented the EdgeStr framework that automatically transforms client-cloud apps into their client-edge-cloud versions. We described and evaluated EdgeStr’s advanced program analysis, profiling, generation, and transformation techniques. Our experiences with applying EdgeStr to representative distributed mobile apps show how it introduces the performance benefits of edge processing, without the high costs of manual program transformation.

ACKNOWLEDGEMENTS

The authors thank the anonymous reviewers, whose insightful comments helped improve this paper. This research is supported by NSF through the grant #2232565.

REFERENCES


