Split and migrate: Resource-driven placement and discovery of microservices at the edge
Genc Tato, Marin Bertier, Etienne Rivière, Cédric Tedeschi

To cite this version:

HAL Id: hal-02401933
https://inria.hal.science/hal-02401933
Submitted on 10 Dec 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers. L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Split and migrate: Resource-driven placement and discovery of microservices at the edge

Genc Tato
Univ Rennes, Inria, CNRS, IRISA, France

Marin Bertier
Univ Rennes, Inria, CNRS, IRISA
INSA Rennes, France

Etienne Rivière
UCLouvain, Belgium

Cédric Tedeschi
Univ Rennes, Inria, CNRS, IRISA, France

Abstract

Microservices architectures combine the use of fine-grained and independently-scalable services with lightweight communication protocols, such as REST calls over HTTP. Microservices bring flexibility to the development and deployment of application back-ends in the cloud.

Applications such as collaborative editing tools require frequent interactions between the front-end running on users’ machines and a back-end formed of multiple microservices. User-perceived latencies depend on their connection to microservices, but also on the interaction patterns between these services and their databases. Placing services at the edge of the network, closer to the users, is necessary to reduce user-perceived latencies. It is however difficult to decide on the placement of complete stateful microservices at one specific core or edge location without trading between a latency reduction for some users and a latency increase for the others.

We present how to dynamically deploy microservices on a combination of core and edge resources to systematically reduce user-perceived latency. Our approach enables the split of stateful microservices, and the placement of the resulting splits on appropriate core and edge sites. Koala, a decentralized and resource-driven service discovery middleware, enables REST calls to reach and use the appropriate split, with only minimal changes to a legacy microservices application. Locality awareness using network coordinates further enables to automatically migrate services split and follow the location of the users. We confirm the effectiveness of our approach with a full prototype and an application to ShareLatex, a microservices-based collaborative editing application.

2012 ACM Subject Classification Information systems → Distributed storage; Information systems → Service discovery and interfaces; Computer systems organization → Cloud computing

Keywords and phrases Distributed applications, Microservices, State management, Edge computing

Introduction

Modern interactive applications combine a front-end running on client devices (e.g. in their web browser) with a back-end in the cloud. Collaborative editing applications, in which multiple users concurrently make changes to the same document, such as Google Docs, Microsoft Office 365, and ShareLatex, are good examples of such interactive applications. Quality of experience for users of such applications depends on low latencies between an action of one client and its visibility by other clients.

A solution to enable fast request-response latencies between the front-end and the back-end of a collaborative application is to deploy part of the back-end at the edge, i.e. on computing resources that are closer and accessible with low latencies from the front-end.
It is often challenging to adapt an application to make use of edge resources. Software monoliths typically require massive re-engineering to support a deployment on multiple sites, as they base the collaboration between their constituents on shared memory or common databases. Service-Oriented Architectures (SOAs) on the other hand present desirable features for this adaptation, by splitting the features of the application into independent services and decoupling service location and naming.

Microservices are a popular approach to SOAs [9, 35] adopted by many large-scale companies [15, 17]. Features of the back-end are handled by fine-grained services communicating through lightweight protocols, such as publish/subscribe or event stores [8]. The most common form of interaction between microservices is the use of point-to-point calls to Representational State Transfer (REST) APIs provided over HTTP.

We are interested in this work in the adaptation of microservices applications towards a joint deployment on core resources, e.g. in some cloud datacenter, and edge resources, e.g. at micro-clouds located in the same metropolitan-area network as the clients. Our objective is to reduce latencies between user actions and their visibility by other users.

We target collaborative editing applications based on microservices. We demonstrated in our previous work [25] that ShareLatex, an open source and microservices-based application for collaboratively editing LaTeX documents, could benefit from reduced user-perceived latencies thanks to a static core/edge deployment of its microservices. This previous work considers however the placement of entire services onto different sites, which may lead to trading latency reduction for some users for latency increases for the others. It also does not consider the adaptation of this placement based on the actual location of the application users.

Contributions

We consider in this paper the dynamic placement and migration of microservices in core/edge deployments. We leverage the use in modern microservices applications of resource-centric REST APIs and NoSQL databases partitioned by a single primary key. This allows us to split microservices, and create independent instances responsible for a partition of the original service’s data. These splits, deployed at different edge locations, can then handle requests for specific partitions of the service data, accessed by close-by users. We demonstrate our ideas with ShareLatex (§2).

Our first contribution is the support for splitting and multi-site placement of microservices. We detail how the state of a microservice can be partitioned, and how the resulting splits can be dynamically deployed on different core and edge sites (§3).

Our second contribution is the middleware support for the decentralized and dynamic discovery of microservice splits. We build on Koala [26], a lightweight Distributed Hash Table (DHT) for decentralized cloud infrastructures. We enable the transparent redirection of calls based on resource identifiers present in HTTP Uniform Resource Identifiers (URIs), also supporting the relocation of microservices splits. This allows adapting compatible legacy microservices applications for hybrid core/edge deployments with minimal effort (§4).

Our third contribution is a locality-driven policy conducting the creation and migration of microservices splits between the core and the edge, and between edge sites themselves, allowing to seamlessly adapt to the location of the users. This policy estimates latencies using network coordinates [13], enabling the automatic selection of the most appropriate site for the services splits used by a group of collaborative users, with the goal of achieving better response times (§5).

We demonstrate our ideas on the ShareLatex application, using a representative core=edge
network topology and measuring the impact of latencies at the level of the application front-end. Our results indicate that Koala and redirection layers induce only minimal overheads, while the dynamic placement of microservices splits enables users in different regions to access the same application with greatly reduced latencies (§6).

Finally, we present related work (§7) and conclude (§8).

2 ShareLatex and its core/edge deployment

ShareLatex is a collaborative application allowing users (e.g. students, researchers or writers of technical documentation) to concurrently edit a LaTeX project. It features a web-based editor with spellchecking and auto-completion, facilities for compiling and producing papers, and tools for the collaboration between writers, such as an address book and a chat service.

Responsiveness is a key element of the perceived quality of service in ShareLatex. For instance, a team of researchers could collaborate on the writing of algorithm pseudocode. Changes made by one researcher must be visible with no undue delay by the others, and changes must propagate as fast as possible to the reference document stored in the back-end to avoid concurrency hazards.

The ShareLatex back-end uses 12 microservices and a database, Redis, shared by four of them (Figure 1). The web provides the front-end to the client browser and acts as an API gateway to other services. User actions (cursor changes, new text, etc.) are propagated by web to the real-time service using WebSockets. The real-time service then sends them to document-updater which is responsible for maintaining a consistent order of modifications. This dynamic state of the project is stored in Redis, and periodic snapshots are pushed to the docstore (text files) and filestore (binary files). Figure 1 details the ShareLatex architecture and its services.

Core servers are typically hosted in a centralized data center, while edge servers are distributed and closer to the users. In our previous work [25], we demonstrated that ShareLatex can benefit from a static placement of its services on a combination of core and edge servers, closer to clients collaborating on a document. We build on our previous contribution, which requires only minimalistic modifications to the configuration and deployment scripts of ShareLatex, and no changes to the application code. The most significant modification

\footnote{Note that using a shared database does not fully comply with the microservices architectural pattern, where all state should be encapsulated in services. Yet, such compromises with the norm are found in many microservices-based applications. We prefer taking them into account rather than heavily modifying the legacy application code.}
performed in our previous work is the disassembly of the web service implementation from its database. This was necessary as web acts as an API gateway and must be deployed at the edge, but it also features a global database of information about users, which is queried infrequently. These queries can be done remotely to a database in the core, with minimal performance penalty.

The static core and edge placement of services of Figure 2 follows the recommendations argued in our previous work [25]: web, real-time, document-updater and Redis should be deployed on an edge site. Due to the coupling of track-changes with Redis, this service must be deployed alongside to avoid remote calls, even if it does not influence perceived latencies as much. The clsi, spelling and chat services can also be deployed at the edge, with a moderate but positive impact on perceived latencies. This placement resulted in lower latencies for operations impacting the most the user experience, at the cost of increasing latencies for operations that require interactions between services at the edge and services remaining in the core.

3 Splitting microservices

While some microservices may be stateless, most of them need to internally store and query data. A stateful microservice is typically implemented as a business-logic tier combined with a database. The choice of the appropriate database is specific to each microservice, leading to what is sometimes called a polyglot architecture. Figure 1 represents the presence of a database inside each service using a small black database symbol. In the unmodified ShareLatex, only real-time is a stateless service. All other services are stateful, including document-updater and track-changes which use the common Redis database. With the decoupling of web from its database (as depicted in Figure 2), this service is also stateless and uses remote calls to a MongoDB service.

A key property of SOA and therefore of microservices is the ability to independently scale in and out the business-logic tier and the database [17]. For the former, new instances may be created and deleted on the fly, e.g. using deployment middleware such as Kubernetes [7] and a scaling policy [28]. Elastic scaling is difficult to realize with relational databases, and microservices state may grow to large sizes requiring the ability to scale out storage to a large number of servers. NoSQL options with such horizontal scaling abilities are therefore a favored choice in many microservices applications.

NoSQL databases such as key/value stores or document stores, partition the data using a unique primary key. We observe that very often, accesses to the database by the business-logic
tier for a query only read and write a limited and identifiable subset of keys. The identification of this subset typically depends on the characteristics of the query, and in particular on its object. It results that the state of the service, i.e. the content of the database, may be partitioned in such a way that keys that are accessed together for any future service requests belong to the same partition. This enables in turn the possibility to create multiple instances of the service, each equipped with one of the partitions. We call these services hosting independent partitions of the database service splits. A service that supports splitting is a splittable service.

Not all services are splittable. Some may require operations (e.g., Map/Reduce queries, scans, etc.) that operate on the entire content of the database. In some cases, it is not possible to identify a mapping between requests characteristics and partitions, e.g. when calls may use object keys generated at runtime or read from the database itself. These services are therefore only replicable: It is only possible to make complete copies of the service and its state. When these copies must be kept in sync for the well-functioning of the application, the service is sync-replicable. When operating on divergent copies does not impact, or impacts only marginally, the well-functioning of the application, provided that users systematically use the same copy, the service is no-sync-replicable.

The analysis of ShareLatex code results in the following categorization of services, also reflected in Figure 2.² The notifications service is sync-replicable, while cls, handling the compilation, is no-sync-replicable: compilations across projects do not require consistent updates. The web service was initially sync-replicable, but the decoupling of its database makes it stateless. All other stateful services—a majority of them—are splittable. This means that their state (content of the services databases, but also the content of the shared Redis database) can be partitioned, and that partitions can be deterministically identified for any query. The object of the query, that allows identifying the partition of service state, and therefore the appropriate service split, is the specific writing project that the user is editing. In other words, the state of ShareLatex splittable services at the bottom of Figure 2 can be partitioned based on the project identifier, resulting in splits able to handle requests for a specific subset of projects. Such splits can then be deployed at the edge, and serve requests from close-by users accessing one of these projects.

The implementation of splitting requires support from the database embedded in splittable microservices, to be able to bulk load and store data partitions between an existing service and a newly created split. This support depends on the database API but does not pose implementation difficulties. For ShareLatex, we built minimalistic APIs enabling this for the Redis and MongoDB databases.

Our goal is to support the dynamic creation of service splits and their deployment over a combination of core and edge resources. This requires both appropriate middleware support mechanisms enabling the discovery and redirection of calls between microservices in a transparent manner, and appropriate adaptation policies to decide at runtime when and where to create splits, and when and where to migrate an existing split if its current location is not optimal. We cover these two aspects in the two following sections.

---

² This identification of services classes and partitions was performed manually, but did not represent a particularly difficult task in the case of ShareLatex. Automated or semi-automated service class identification and partitioning are beyond the scope of this paper, but we intend to explore these directions in our future work.
We now present the mechanisms that support the dynamic deployment of service splits on multiple sites. Our focus in this section is on the proper functioning of the system during and after service splitting and migration operations. We present the policies triggering these operations in the next section.

Our support middleware serves two purposes: Firstly, it enables the discovery of services and splits, and the live modification of their placement (§4.1). Secondly, it enables the redirection of point-to-point calls between source and destination services, ensuring that the core service or its appropriate split is reached (§4.2).

### 4.1 Discovery of microservice splits with Koala

Each service is initially associated with one instance in the core (the core service), responsible for its full state. Split and migrate operations dynamically update the list of splits for each service. Service discovery, therefore, requires the maintenance of an index of existing services, together with their current lists of splits. Every such split is associated with a list of object identifiers, for which this split is the only one able to process queries. This index must remain strongly consistent: At any point in time, there must be a single core service or split that can answer a query for a given object, and it must be impossible for two clients of the service under the same object to use different splits concurrently.

Service registries based on replicated databases updated using consensus (e.g., using etcd [11] or ZooKeeper [18]) are adapted for datacenter deployments with low network latencies. In our target context of distributed sites, centralizing the index would result in unacceptable overheads. We favor instead a decentralized design, supporting the caching and lazy revocation of split-to-site associations. This service is distributed, with an instance running at the core and at each of the edge sites.

Service discovery requests contain the name of the service, and for splittable services, the identifier of the query object. For ShareLatex splittable services, this object is the project identifier, that allows identifying the appropriate service state partition. Service discovery requests can be addressed to any of the sites.

The service index is implemented as a Distributed Hash Table (DHT), in which each node stores a subset of the index, partitioned using consistent hashing. Index elements are accessed using a primary key. Each node is responsible for a range of these keys. An overlay enables requests to deterministically reach the responsible node using greedy routing (each node in the path selects amongst the nodes it knows the closest to the destination). Typical DHT designs actively maintain all overlay links through the exchange of explicit overlay construction messages. In this work, we rely on Koala [27], a DHT that creates overlay links in a lazy manner, by piggybacking overlay construction messages over existing application traffic. This design choice enables to create more overlay links for routes in the overlay that are more frequently used for index reading requests, and minimize maintenance costs for seldom-used links. This is beneficial for workloads that are highly local, which is expected from service requests in one single application and to a relatively limited number of services (e.g. up to a few hundred).

### Indexing

We keep two global indexes in Koala, an index of Objects, and an index of Splits. Figure 3 shows an example of the local subset of these indexes maintained by one Koala node. A Koala
node is responsible for maintaining the authoritative and strongly consistent entry for a number of index items, falling in its key responsibility range. It also maintains local resources, objects and splits, that are hosted on the corresponding edge site. A Koala node may have local resources for which it is not responsible or be responsible for resources that are not local. This design enables the creation of resources on a different node than the one that the DHT assigns for the corresponding entry index, while maintaining a single node in charge of this index entry and allowing atomic modifications. Lookups follow multiple hops in the overlay, until the responsible node is found, leading to one last hop to the node where the entry is local (if different). Nodes hosting locally a resource access it without involving the responsible node.

**Discovery**

A local split can only be reached by proxying through the local Koala instance. The discovery of the appropriate site for an incoming service request proceeds in two phases. First, the Objects table in the DHT is queried to establish whether there exists a split of at least one service under that object. This information is stored in the split group for that object. If there is no entry for the object, or if there is no entry for the specific service in the split group, the request must go to the core. Second, the Koala node responsible for the split is located using the Splits table, using both the service name and split number as the key. This requires reaching the Koala node that is responsible for that key and then reaching the Koala node where that split is local.

For instance, on the node whose local subset of the index is represented by Figure 3, a request to Service 2 for Object 1 will be immediately sent to Service 2’s Split 1 hosted locally. A request for Object 3 will be redirected in one hop to Koala node of identifier 5-2, to read its split group. A request for Object 4, not present in the local state, requires a multi-hop routing in the Koala overlay to request its service group.

**Caching**

Looking up service discovery entries in the DHT for every service call is too expensive. We implement caching: results of index lookup are kept at the local Koala node and reused. Stale cache entries are discarded in a lazy fashion. We leverage the fact that all requests must go through the local Koala node, e.g. on the edge site where the split actually runs. After the migration to a new site, queries based on stale cached information will arrive at the Koala node at the previous local location of the split. This node simply informs the origin, which invalidates related cache entries and falls back to a regular lookup.

---

3 Allowing uncontrolled connections from outside of the edge site might not be possible due to network constraints, or not desirable for security reasons. The local Koala node acts, therefore, as an API gateway for all local service splits.
split and migrate

The migration of an existing split, or the creation of a new split, follows four phases. Firstly, an instance of the service is bootstrapped if none already exists at the destination edge site, or it is selected among existing instances, but it does not hold state or service requests. Secondly, a new entry in the Splits table is created to announce the existence of the new split. It does not contain a location yet. The split group for all corresponding objects is updated to indicate the temporary unavailability of the split. Service requests will block at the lookup request stage, and back off for a random time duration. Thirdly, the new instance receives the partition of the data from the source service or split. Finally, the Koala entry for the split is updated to reflect the location of the new local site for that split, and the split groups for all corresponding objects are updated. This allows request services to resume, using the new split location.

4.2 Transparent redirection of REST service calls

Modifying legacy microservices applications to directly make use of Koala APIs to discover and call services and splits would require an important effort. Instead, we leverage the fact that the objects of queries are accessible in the URIs of REST service calls. Indeed, REST being a resource-centric approach to designing interfaces, calls are made, typically over HTTP, to an explicit resource given in the request URI. We implement the transparent redirection of calls by extracting the object from this URI. Then, the local Koala node queries for the existence of a split for that object and the requested service. The request URI is transformed using rewriting rules to reach either the original core service, or the Koala node on the edge site where the split runs.

The implementation of the redirection is as follows. It is illustrated for a call in ShareLatex in Figure 4. We use the high-performance web server nginx as a reverse proxy for calls from, and to, local services. In ShareLatex, this includes the web service that serves as an API gateway for the user frontend. The reverse proxy translates the original request from the unmodified ShareLatex, to a request to the local Koala node. The discovery process detailed before establishes that there exists a split for that service that must serve the request. In the example of Figure 4, the web service on the Edge 1 site calls the chat service. The object “123”, the project identifier, is extracted from the call URI. Koala then determines that the service split is on the Edge 2 site. The request is redirected to chat service in that site, where the call is handled by Koala.
The creation of service splits and their migration between sites obey an adaptation policy. This policy must determine what service to split, when these split decisions are made and where to (re)deploy the splits. Its goal is to ensure that user-perceived latencies in the application are minimized.

**What service to split?**

The first aspect of the policy is application-dependent and results from the analysis of the interactions between its microservices. A set of splittable services, and not necessarily all of them, must be tagged for a preferential deployment at the edge. This aspect of the ShareLatex policy builds upon our previous results [25] (§2). Microservices that lie in the bottom part of Figure 2 are tagged for edge deployment. All other services always remain in the core.

**When should splits happen?**

There are two situations where a split may be formed: When a new object is created, and when latencies to the core are too high. The first option is sufficient for the ShareLatex policy: The creation of a new project leads to the immediate creation of all corresponding splits.

**Where should splits go?**

This aspect of the policy is twofold: Firstly, we must ensure that splits are created on a site (core or edge) close to the first user of the corresponding object. Secondly, we must adapt this placement when the chosen site is no longer the most adequate for the current set of users of that object.

This requires the ability to evaluate network latencies. Active probing of latencies (e.g. using ICMP packets) is impractical and unscalable. We combine two mechanisms to enable probe-less estimations. Firstly, we enforce that users always connect to the geographically closest site. The location of a client is that of its connection site. Secondly, latencies between sites are estimated using Network Coordinates (NCs). We use Vivaldi [13] to compute NCs. Each site is represented by a d-dimensional point. These points positions evolve following a process similar to a spring-mass relaxation, based on observed latencies for actual message exchanges, and Euclidean distances eventually approximate latencies.

The ShareLatex policy enforces that the initial version of an object, and the corresponding splits, be hosted by the connection site of the first user. Each site collects for its local splits, a history of the NCs of the sites forwarding client calls. Periodically (every 5 minutes, or 100 requests, whichever comes first, in our implementation), the policy determines whether migration of the splits for each hosted object is necessary. Several users access a project, from different sites and with different frequencies. The ideal location of the splits for that project can be represented as a point in the NCs space. We define this point as the Center of Mass (CoM) for that object. It is the geometric average of the connection sites’ NCs, weighted by the number of accesses from their clients. If there exists a site whose NC is closer to the CoM, the policy triggers a migration of all splits for that object to this new site.

---

4 The list of core and edges sites IP is publicly known. Clients use an IP-to-location service (e.g. www.iplocation.net) and choose the geographically closest site.
6 Evaluation

We evaluate the split and migrate principles with a full prototype, combining Koala, nginx reverse proxies, Docker CE for bootstrapping containers on the core and edge sites, and ShareLatex as the application.

Our evaluation aims at answering the following research questions: (i) Is the approach able to reduce perceived latencies for users of the application? (ii) Can the policy successfully migrate splits between edge sites when users’ locations change? (iii) Is the overhead of using Koala and proxying acceptable?

We consider the three-layer (L1-L3) hierarchical topology shown in Figure 5. Its characteristics are derived from information obtained from an Internet Service Provider in the EU [23]. Layer L1 consists of the core site, L2 of regional sites (reg1) and L3 of edge sites (edge1, edge2 and edge3). We deploy each site on a node of the Grid’5000 [5] testbed. Each node features 2 Intel Xeon E5-2630 v3 CPUs and 128GB of RAM. We emulate latencies between sites using the tc (traffic control) tool. Note that reg1 is treated as an edge site, and that we ignore latencies between users and sites, and model their mobility by enforcing that they connect to a specific (closest) site. We use Network Coordinates (NCs) in $d = 2$ dimensions for ease of presentation, although a higher dimensionality (e.g. $d = 5$) would yield better estimations. Latencies are measured at the level of the instrumented ShareLatex frontend. We emulate the activity of users using the Locust [1] load testing tool, which allows describing programmatically the behavior of users as a list of actions and their respective occurrence frequencies.

6.1 Adaptation and split migrations for moving users

Our first experiment evaluates the ability of our approach to adapt the location of the splits for single a ShareLatex project, and the impact this has on latencies. We consider a project $p$ shared by two equally active users, one stationary and one who changes her location continuously. Each user performs one operation every second, adding a new character to the text. The user-perceived latency is measured from the moment the text is updated by one user to the moment the update appears in the screen of the other user.

Figure 5 presents the experiment setup. Figure 6 presents the evolution of the average perceived latency for the two users, and Figure 7 presents the evolution of the CoM of the project. Circled numbers in all figures show the sequence of operations.

We follow three phases. In each phase, users are assigned to connection sites, and we observe the triggering and impact of the adaptation and resulting split migration decisions. Initially, both users are closer to edge1 and therefore connect to that site. The latency for updating the text (50 ms) is roughly the RTT between edge1 and core, plus the processing
time, of 40 ms and 10 ms respectively (3 in Figure 6). Given that all requests for project \( p \) originate from the Koala instance on edge1, that location is also the CoM (3 in Figure 7), and therefore the policy decides to split and migrate all tagged services to this site (2). The latency drops to slightly over the processing time. In a second phase, we move one of the users to edge2 while the service splits for the project are still in edge1 (3). This results in an increase in latencies. When it next triggers, the adaptation policy decides to migrate the service splits to reg1 which is closer to the new CoM for the project (4). In the third phase, we move the user of edge2 and connect it to edge3 (4). The service splits are still in reg1, which results in high latencies. Again, the adaptation policy triggers and orders the migration of splits to the closest site to the CoM (5). The core happens to be the best compromise to serve the two users connected to edge1 and edge3. This experiment shows that the policy is effective in splitting and migrating a single project according to its user locations, for a positive impact on perceived latencies.

### Table 1 Distribution of projects, users and ideal site placements.

<table>
<thead>
<tr>
<th>Project</th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>p4</th>
<th>p5</th>
<th>p6</th>
<th>p7</th>
<th>p8</th>
<th>p9</th>
<th>p10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Users</td>
<td>u1, u5, u6</td>
<td>u2</td>
<td>u1, u3</td>
<td>u1, u4</td>
<td>u5, u6</td>
<td>u6</td>
<td>u4, u7</td>
<td>u5, u8, u9</td>
<td>u5, u8, u9</td>
<td>u8, u10</td>
</tr>
<tr>
<td>User locations</td>
<td>e1, e2</td>
<td>e1</td>
<td>e1, e1</td>
<td>e1, e1</td>
<td>e2, e2</td>
<td>e2</td>
<td>e1, e2</td>
<td>e2, e3, e3</td>
<td>e2, e3, e3</td>
<td>e3, e3</td>
</tr>
<tr>
<td>Ideal site(s)</td>
<td>e2, r1, e1</td>
<td>e1</td>
<td>e1</td>
<td>e1</td>
<td>e1</td>
<td>e2</td>
<td>e2, r1, e1, e2</td>
<td>e3, core, e2</td>
<td>e3, core, e2</td>
<td>e3</td>
</tr>
</tbody>
</table>

### 6.2 Evolution of splits distributions

This second experiment shows how the split and migrate principles allow shifting the load from the core servers to edge servers while following the location of the most active users in a collection of ShareLaTeX projects. All services are initially only in core. We consider 10 users and 10 projects. Each project is edited by 1, 2 or 3 users. The two first lines of Table 1 show the mapping between users and projects. The third line indicates the (static) user locations for each project.

We model the activity of users to represent work sessions. During one hour and a half, every user randomly picks one of their assigned projects and edits if for a random duration of 2 to 10 minutes. The project CoM evokes to follow the location(s) of the currently active user(s). The fourth line of Table 1 indicates the possible ideal location(s) for the project splits, calculated offline.

We monitor the location of the service splits for the different projects, taking snapshots
every 1,000 seconds. We run this experiment until the projects with a single ideal site placement reach this destination. Figure 8 presents these snapshots and the location of the service slices for the 10 projects. Projects whose ideal site is unique, such as p2-p5 and p10, have the corresponding service slices migrated to these sites correctly and immediately. Projects with multiple ideal sites see their slices periodically migrate between these sites, following the currently active user(s). For instance, splits for p7 move between reg1 and edge2, while splits for p8 and p9 move between edge3 and core. The final site is highlighted in boldface in Table 1. This experiment shows that the split and migrate mechanisms and the adaptation policy for ShareLatex allow dynamically moving microservices close to the users, based on the used resources (projects in ShareLatex).

6.3 Overheads of Koala and redirections

In this final experiment we evaluate the costs and overheads of the mechanisms enabling transparent call redirections. To isolate the overhead we compare a centralized setting where everything is deployed in the core, corresponding to the original ShareLatex model, with a one-edge-site setting where requests are redirected from this edge site to the core by Koala. Figure 9 presents this setup. We use a 50 ms latency between edge and core sites.

In both settings, the service split that responds to the user request is in core. In the centralized setting the request is first sent to the web core service and then forwarded to the right service directly, while in the second setting the request goes first through the local web split. This proxies the request to the Koala instance on edge1, which in turn forwards it to the Koala instance in core who then calls the service.

We distinguish three kinds of requests, two HTTP REST calls and one WebSocket request. For the REST calls, we consider a call to tags, for which splitting is disallowed (①), and a call to chat, which is splittable using the project identifier as the object (②). The WebSocket request updates the text (writing) ③. It is also a project-specific request and must reach the corresponding split of the document-updater service.

We expect a slightly higher overhead for redirections to split services compared to non-split ones. For non-split services, a single interaction with Koala is required (follow ①). For split services, two interactions are necessary: one to locate the object and one to redirect to the correct split (follow ② and ③).

The operation latencies times of the three requests with and without the redirection are shown in Figure 10. We consider two cases for the redirection: without and with caching. When the cache is disabled, lookups on the Koala DHT can require multiple hops between
sites and incur a significant and unpredictable penalty. With caching, this penalty is only paid for the first access or after a migration invalidates the cached information. WebSockets requests occur on an established connection, therefore caching does not apply.

Figure 10 presents the distribution of latencies for the three operations and for 500 requests each. We observe a similar performance between the centralized setting and the setup using caching. The median overhead of proxying through the local edge site is \( \approx 3 \text{ ms} \) for the non-split service and \( \approx 4 \text{ ms} \) for the split one. For WebSockets operations this difference is smaller, \( \approx 1 \text{ ms} \), which can be explained by the fact that this protocol is more lightweight than HTTP. Disabling caching leads to significant overheads as every operation leads to lookups in the DHT, bouncing between the core and edge Koala instances. This experiment shows that the latency impact of proxying through the edge is likely to be negligible compared to the gain of using locally-deployed services splits.

### 7 Related work

Previous research advocates to revisit the SOA paradigm for supporting service-based applications deployed in edge cloud platforms [19]: In light of the increase of the number of services at the edge able to answer a specific query, service registration must take into account spatial coverage, and service discovery must take locality into account. Our contributions are a step in that direction.

The placement of applications on fog platforms has been an active research topic in the
recent years. One target domain is IoT applications where data collected from connected objects must be processed on nearby resources [24, 34]. Stream processing is another application that benefits from deployments on a combination of core and edge resources. It explicits its communication patterns (i.e., the directed acyclic graph linking stream processing operators), which can be leveraged for optimal placement on edge resources [12]. The Balanced RePartitioning (BRP) [4] algorithm targets generic distributed cloud applications and devises online algorithms which find a good trade-off between communication and migration costs.

Our work is linked with the concept of mobile edge clouds, where users move and connect to nearby resources dynamically [30]. When the mobility of users is modeled using Markov stochastic decision processes, analytical frameworks allow devising close-to-optimal algorithms for automating service placement [31]. Other approaches advocate the use of genetic algorithms to gradually refine an allocation of services to the edge [33].

We note that all of the aforementioned work considers the placement (and in some cases the migration) of full instances of services. We are not aware of solutions proposing to split stateful microservices and support resource-based discovery. State splitting is used, in a different context, for the elastic scaling of publish/subscribe middleware [6].

Research on collaborative edition has focused on enabling correctness and performance, including in the presence of network issues. The Jupiter protocol [21, 32] and the RGA protocol [22] implement a replicated list object abstraction and define how to propagate updates to achieve convergence [3]. Our work is complementary: The responsiveness of replicated list object algorithms (i.e. the time between an update and its visibility at the other clients) is sensitive to the latency between client nodes and a coordination server.

Service discovery middleware solutions for data centers typically rely on strongly consistent, fully replicated stores maintaining the complete index of services instances and of their locations. SmartStack [2], used for example by the Synapse [29] microservices platform, is based on Apache ZooKeeper [18]. Similarly to Koala, Synapse instances provide local proxies to services, but each maintains a full copy of the index while Koala relies on a DHT and caching for scalability. Kubernetes [10] leverages etcd [11] for service discovery. Recent work [14] suggests to add support for network coordinates [13] to route requests based on network locality. Yet, service selection decision remains a centralized process unlike with Koala where it can happen at the edge. Eureka [20] is also centralized but introduces the notion of read clusters that can serve requests closer to the clients. Unlike lazy cache management in Koala, read clusters must be explicitly synchronized when the service index changes. Write clusters can also be replicated, but are only eventually consistent, which makes them ill-suited for implementing consistent service migration. Finally, Consul [16] supports deployment to multiple data centers, and use network coordinates for location-aware selection. Consul only uses consensus-based synchronization within each individual data center. Updates propagate lazily between data centers using gossip, preventing consistent service relocation across data centers.

8 Conclusion

We presented how microservices could be dynamically deployed on a combination of core and edge resources. Our approach leverages the possibility to split microservices for which partitions of the data can be used to answer subsets of service requests independently. The Koala middleware enables to transparently redirect requests to the appropriate split based on object information available in REST calls URIs. Migration policies enable a dynamic placement of microservices splits on edge sites, and as our evaluation with the ShareLatex application shows, allow following the users and reduce perceived latencies.
This work opens interesting perspectives that we intend to consider in our future work. First, we wish to explore the automation of the identification of splittable microservices, and the use of static and dynamic analysis techniques to infer the relation between objects and state partitions. Second, we intend to extend support middleware to support redirections with other forms of communication, such as publish/subscribe or event sourcing [8]. Finally, we would like to build tools to automatize the identification of placement policies based on dynamic observations of communications between microservices.

Acknowledgments
We thank the anonymous reviewers for their comments. This work was partially funded by the Belgian FNRS project DAPOCA (33694591) and partly supported by the Inria Project Lab program Discovery (http://beyondtheclouds.github.io/).

References
11 CoreOS. Etcd reliable key-value store. https://coreos.com/etcd/.
29:16 Split and migrate


