Handling Memory Pointers in Communication between Microservices

1st Vini Kanvar
IBM Research, India
vkanv031@in.ibm.com

2nd Srikanth Tamilselvam
IBM Research, India
srikanth.tamilselvam@in.ibm.com

3rd Raghavan Komondoor
Indian Institute of Science, India
raghavan@iisc.ac.in

***Abstract*—When microservices are written from scratch, APIs are usually made stateless. However, when an existing monolith application is decomposed into microservices, it may not be possible to make all the APIs stateless. Therefore, objects transferred via APIs may contain pointers. Consequently, data transfer via an API i.e., from a client address space to a server address space, reconstruction at the server, and returning to the client become non-trivial operations.**

Conventionally, data transfer between microservices is done using JSON, which serializes pointers to values that they point to. Once the data in JSON reaches the server, deserialization creates objects of the original types on the server. However, deserialization is unable to return the same objects passed by the client because serialization leads to loss of pointer information. We propose to apply *pointer swizzling* to solve this problem. Pointer swizzling modifies the definition of the class by introducing ID of the object and by replacing all pointers with IDs of the objects it refers. These IDs help to maintain correct reference in the server. After the server API operates on the objects, the server returns new objects of the same types to the client. These new objects need to be *plugged* back in the client address space i.e., pointers to the old objects in the client need to now point to the corresponding new objects. This plugging back is non-trivial because we do not know how the old objects map to the new objects. We propose creating *memory maps* at runtime to overcome this challenge.

***Index Terms*—pointers, API, monolith, refactoring, swizzling**

I. Introduction

Automating decomposition of monolith into microservices is an active area of research [1]–[5]. These work perform static code analysis or use runtime logs to infer the implementation structure and represent the monolith as a graph where all the programs become nodes and their dependencies with other programs become edges. The decomposition task can thus be deduced as a graph clustering task. To achieve functional independence, they cluster programs in such a way that connections within the clusters are maximised and connections across clusters are minimized. Therefore, each cluster can be considered as a candidate microservice. But none of these work discuss how data should be communicated via APIs between microservices.

Memory address spaces of such decomposed microservices are not shared. Therefore, we cannot pass objects as it is via APIs. Usually, JSON of each object is passed between microservices via the APIs. Objects with pointers when transferred in JSON format, lose information. In the presence of pointers, JSON serializes pointers to values that they point to.

```
class OrderDataBean {
    HoldingDataBean holding;
    AccountDataBean account;
    ...
}

class HoldingDataBean {
  AccountDataBean account;
  ...
}

class AccountDataBean {
  private BigDecimal balance;
  public void setBalance(BigDecimal balance) {
      this.balance = balance;}
  ...
}

class TradeSLSBBean {
  public OrderDataBean buy(...) {
    ...
    AccountDataBean account = ...;
    // createOrder(.) sets order.account = account
    OrderDataBean order = createOrder(account,...);
    account.setBalance(balance);
    ...
    return order;
  }

  public OrderDataBean completeOrder(...) {
    ...
    OrderDataBean order = ...;
    AccountDataBean account = order.getAccount();
    HoldingDataBean newHolding = ...;
    // order.account and newHolding.account are
    // aliased to account
    ...
    order.setHolding(newHolding);
    ...
    return order;
  }
```

Fig. 1. Code snippets from DayTrader application to motivate the need of handling pointers and aliases in communication between microservices. Refactored code in Figure 2 shows how to handle the communication.

Two microservices can be thought of acting as a client and a server, if one calls the APIs of the other, respectively. The communication via APIs needs to handle the following:

- Accurate transfer of objects containing pointers between client and server address space.
- Plugging i.e., copying objects from server back to the client address space to update the state.

```
class TradeSLSBBean {
  public OrderDataBean buy(...) {
    ...
    AccountDataBean account = ...;
    // createOrder(.) sets order.account = account
    OrderDataBean order = createOrder(account,...);

    // Refactor the method call to REST API call
    // account.setBalance(balance);

    String url="http://<host>:<port>/accountdatabean/setbalance";
    RestTemplate restTemplate = new RestTemplate();
    AccountRequest request = new AccountRequest(account, balance);
    AccountDataBean newAccount =
      restTemplate.postForObject(url, request, AccountDataBean.class);

    // newAccount should replace account. This motivates the need
    // of handling aliases (e.g. between order.account and account).
    Map<AccountDataBean, AccountDataBean> memoryMap =
      Algorithm1(account, newAccount);
    Algorithm2(memoryMap);

    ...
    return order;
  }
```

```
public OrderDataBean completeOrder(...) {
  ...
  OrderDataBean order = ...;
  AccountDataBean account = order.getAccount();
  HoldingDataBean newHolding = ...;
  // order.account and newHolding.account are aliased to account.
  ...
  // Refactor the method call to REST API call
  // order.setHolding(newHolding);

  String url="http://<host>:<port>/orderdatabean/setholding";
  RestTemplate restTemplate = new RestTemplate();
  AccountRequest request = new AccountRequest(order, newHolding);
  // The API call sends JSON of request. The alias between
  // order.account and newHolding.account should be maintained.
  // This motivates the need of pointer swizzling.
  OrderDataBean newOrder =
    restTemplate.postForObject(url, request, OrderDataBean.class);

  Map<OrderDataBean,OrderDataBean> memoryMap =
    Algorithm1(order, newOrder);
  Algorithm2(memoryMap);
  ...
  return order;
}
```

Fig. 2. Refactoring of class TradeSLSBBean of Figure 1 to show how pointers and aliases are handled in communication via APIs between microservices. Algorithms 1 and 2 are proposed by us to handle pointers in microservices architecture.

To motivate the need of solving this problem, we show code snippets from a public application, DayTrader [6] in Figure 1. The application emulates an online stock trading system. It is written as a monolith i.e., it is a single deployable unit. It contains a class OrderDataBean which holds a pointer to class HoldingDataBean using field holding. Both these classes hold pointers to class AccountDataBean using the field account. Class AccountDataBean stores the balance information. Class TradeSLSBBean contains methods buy(.) and completeOrder(.), which are invoked from the user interface to update the balance information. Objects order, account, newHolding are created in these methods. Field account of order is accessed using a dot notation as order.account in Java. Method buy(.) makes order.account point to object account. It calls setBalance(.) of AccountDataBean. Method completeOrder(.) makes order.account and newHolding.account point to object account. It calls setHolding(.) of OrderDataBean. Updated order is returned by the methods.

Existing clustering techniques [1] recommend class TradeSLSBBean in one cluster and AccountDataBean, HoldingDataBean, and OrderDataBean classes in another cluster because of the functional properties of the classes. The affinity i.e., edges between the classes across these two clusters are less compared to the ones within the cluster. Each of the cluster therefore becomes a candidate microservice supporting a common functionality through the classes it contains. Although the clustering algorithm works with an objective function to be self dependent, dependency with other microservices cannot be completely avoided and might be required to complete a business function. Therefore, communication between the two microservices needs to be happen via APIs. In this case setBalance(.) and setHolding(.) methods needs to be converted to APIs.

Due to the presence of pointers and aliases, we need to ensure the following:

- Accurate transfer of objects order and newHolding containing pointers via API setHolding(.) of OrderDataBean.
- Plugging i.e., copying object account from server microservice containing AccountDataBean back to the client address space of microservice containing TradeSLSBBean. Object account is updated by API setBalance(.) of AccountDataBean.

Figure 2 also shows method completeOrder(.) makes order.account and newHolding.account point to object account. It creates a REST API call to setHolding(.) and passes object order and newHolding. API postForObject(.) that transfers information, serializes objects into JSON. However, JSON does not preserve alias information. It loses that order.account and newHolding.account are aliased. Therefore, we propose to override postForObject(.) that transfers pointer swizzled JSON to preserve the aliases during transfer.

Serialization into JSON loses alias information. We solve the problem of inaccurate serialization using *pointer swizzling*. Instead of serializing the pointers to values, it creates ids of pointers; thereby maintaining alias information.

Figure 2 shows how refactored methods setBalance(.) and setHolding(.) have been exposed as APIs and how pointers are communicated between the microservices. Method buy(.) makes order.account point to object account. It creates a REST API call to setBalance(.) and passes object account and balance information. The API returns newAccount containing the updated account. We need to plug newAccount from server back to account in the client. In other words, we need to set account = newAccount and we need to set all existing aliases i.e., order.account = newAccount. This plugging of returned objects in the client is a non-trivial task because we do not know how the old objects map to the returned objects. Any static alias analysis is imprecise [7] and may not be able to discover this. Therefore, we propose to use *memory map* to plug newAccount back to account.

A *memory map* is a map between the old and the returned group of linked objects. This allows to copy information from

the returned objects to the old objects, delete the old objects that got deleted by the server API, and create objects that were created by the server API.

We assign a unique global ID to each object; the ID could be formed by concatenating the owning microservice name of the object with an ID that is unique within that microservice. We need the IDs to be globally unique for memory map; they need not be globally unique for pointer swizzling. However, since we any way maintain globally unique IDs for memory map, we may as well use the same IDs for pointer swizzling.

The rest of the paper is organized as follows. The inaccuracy of serialization and its solution using pointer swizzling is presented in Section II. The need of plugging back and creating memory maps is presented in Section III. Experiments and empirical measurements are in Section IV. Related work is in Section V. We conclude the paper with future work in Section VI.

II. Serialization and Deserialization

JSON (JavaScript Object Notation) is a format commonly used to exchange data between microservices. It consists of attribute–value pairs. If the data contains pointers, the pointers are serialized into values that they point to. An example of serialization of objects is shown in Figure 3(a). In this section, we show the inaccuracy in serialization of pointers and we solve this using pointer swizzling.

We represent memory using graphs [8] where nodes denote memory locations and edges denote memory links representing the address of a target location stored in a pointer. Edges are of two types: (i) unlabeled edges from variables like obj to nodes and (ii) labeled edges from nodes to nodes, whose label is the field name that stores the pointer.

A. Loss of Pointer Information During Serialization

JSON replaces pointers with values of the objects they point to; it does not save the addresses of the pointers. Therefore, alias information is lost. Figure 3(a) shows this problem. Structure pointed by obj is shown; value of field n is written inside nodes. The structure has an alias where obj.f.f and obj.g.f point to the same node ”d”. The JSON i.e., serialization of obj contains {”n”:”d”,...} twice. Therefore, when this JSON is deserialized on the server, it cannot be determined whether {”n”:”d”,...} appearing twice, refers to the same object or not. Deserialization inaccurately creates two objects for this JSON.

B. Solution: Pointer Swizzling

Pointer swizzling/unswizzling [9], [10] is a technique to deserialize/serialize, respectively. Pointer unswizzling replaces pointers with ids, and pointer swizzling does the reverse.

This technique is used in object-databases. When an in-memory object is serialized and persisted, the pointers in it are converted to unique IDs. When the object is deserialized and loaded back to memory from the persistent store, the IDs are *swizzled* back to normal pointers. The communication channel between the microservices is analogous to the database. Figure 3(b) shows how pointer swizzling solves the problem on our example. Pointer swizzling modifies the definition of class T by including ID of the object and by replacing all pointers with IDs of the target objects viz. fID, gID. The graph in the middle denotes ID, fID in superscript, gID in subscript of each node. Edges denote pointer unswizzled references. Nodes with ID 2 and 3 both have fID as 4, which accurately denotes the alias. Therefore, serialization does not lose any information and deserialization on server obtains the same information sent by the client.

Advantages of pointer unswizzling over JSON serialization:

- JSON serialization of recursive data structures throws exception. However, pointer unswizzling allows this serialization.
- In the presence of aliases, pointer swizzled JSON is more efficient to compute. Without pointer swizzling, JSON requires serialization of all pointers for each path in the graph.

III. Plugging Back

In this section, we explain the problem of plugging back and give its solution using memory maps.

A. Loss of Alias Information After State Update

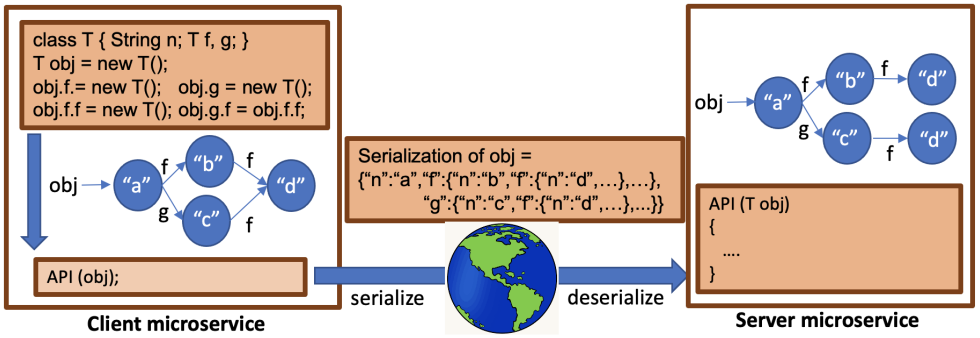
If a parameter of an API represents an object containing pointers, the client needs to pass the group of linked objects reachable from the parameter to the server. However, external pointers i.e., objects that point externally to the objects, are not passed to the server. After the server API performs its operations on the group of linked objects, it returns these to the client. The client needs to replace the old objects with the updated objects returned by the server. Further we need to update the external pointers i.e., objects at the client that pointed to the old objects and were not passed to the server. These external pointers need to now point to the corresponding returned objects. This plugging of the returned objects in the client is a non-trivial task because we do not know how the old objects map to the returned objects. Figure 4(a) shows the problem of plugging external pointers x and y.f to the new objects on return to the client.

In Figure 4(a), structure pointed by parameter obj before API call on client microservice is shown with dotted lines. This is passed to the server. Structure obtained after API operations is shown after API on server microservice. After returning the structure to the client, it is unknown how to plug back i.e., which nodes should x and y.f now point to? Two possible structures are shown after API on client microservice.

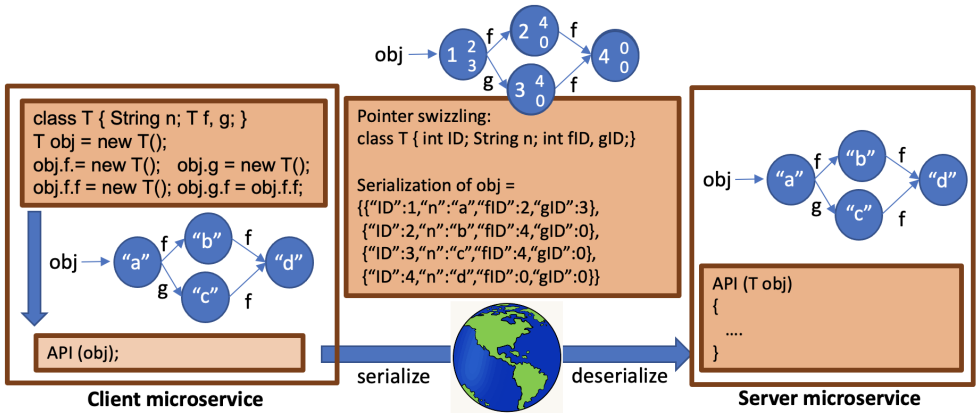
In order to map the old object with the new object obtained from deserialization of JSON, we propose to create a memory map.

B. Solution: Memory Map

Since static pointer analysis [7] is inaccurate in mapping old objects with the new objects, we perform the mapping at runtime. For this, we add an ID field to each object and replace pointers with IDs using pointer unswizzling (Section II-B). We assign a unique global ID to each object; the ID could be



(a) Problem with serialization of structures containing aliases.



(b) Pointer swizzling to transfer objects containing pointers.

Fig. 3. Problem with serialization of structures containing aliases is shown in (a) which is solved using pointer swizzling in (b).

Algorithm 1 Map old and new group of linked objects

Input: Roots of old and new group of linked objects, respectively
Output: Map of each old object to its new object
Procedure computeMap (*oldRoot*, *newRoot*)
  $\forall$ *old* $\in$ objects reachable from *oldRoot*,
    $\exists$ *new* $\in$ objects reachable from *newRoot*,
      s.t. *old.ID* = *new.ID* $\iff$ *(old,new)* $\in$ *memoryMap*
  $\forall$ *old* $\in$ objects reachable from *oldRoot*,
    $\nexists$ *new* $\in$ objects reachable from *newRoot*,
      s.t. *old.ID* = *new.ID* $\iff$ *(old,*ϕ*)* $\in$ *memoryMap*
  $\forall$ *new* $\in$ objects reachable from *newRoot*,
    $\nexists$ *old* $\in$ objects reachable from *oldRoot*,
      s.t. *old.ID* = *new.ID* $\iff$ *(*ϕ*,new)* $\in$ *memoryMap*
  return *memoryMap*

formed by concatenating the owning microservice name of the object with an ID that is unique within that microservice. With this, when the new objects are returned to the client, their IDs can help us map the old objects with the returned objects. We copy the fields of each returned object to the old object, and delete the returned object on the client. If any object contains a new ID that did not exist before the API call, then it can be determined that the object has been newly created by the server. Such an object should not be deleted on the client.

We discuss the approach using algorithms 1 and 2. Algorithm 1 takes the old group of linked objects on the client before the API call and the new group of linked objects received from the server, and maps the two. Algorithm 2 then copies the fields of each new object to the old object, and

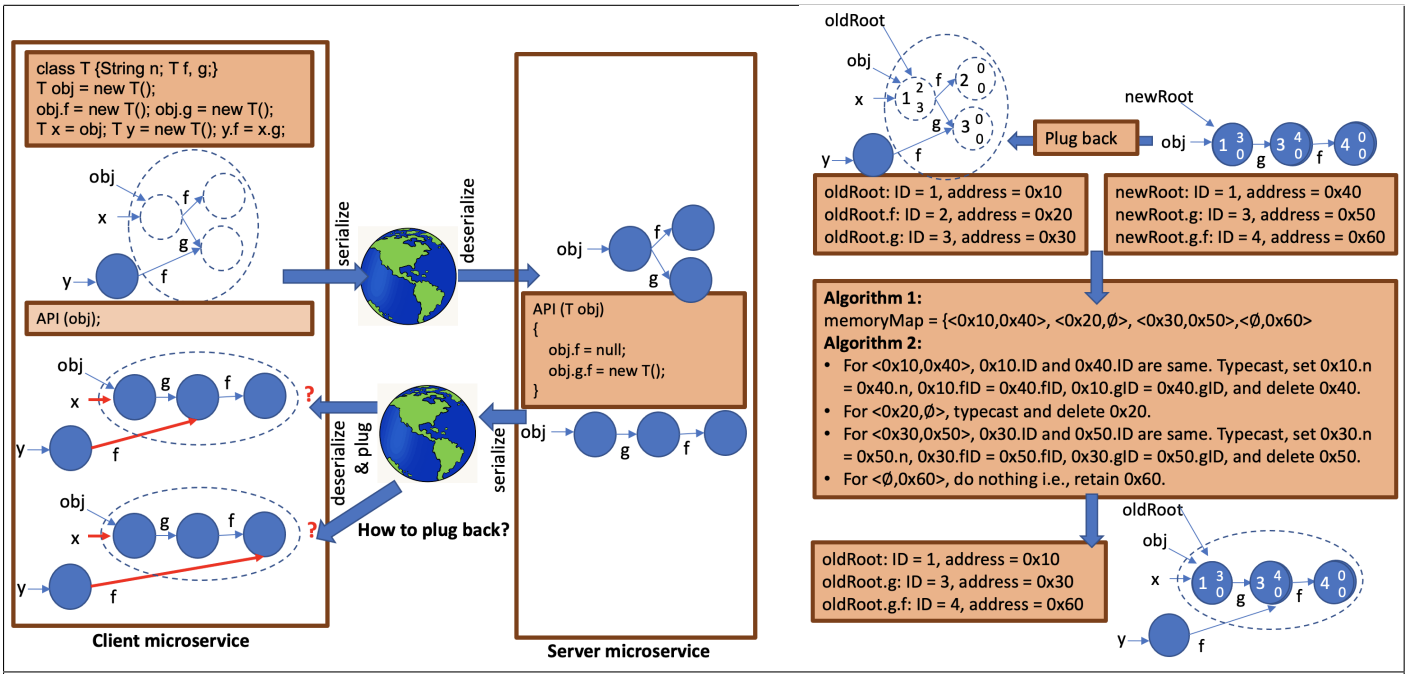
Algorithm 2 Update old objects with mapped new objects

Input: Map of each old object to its new object (from Algorithm 1)
Procedure updateOldObjects (*memoryMap*)
  $\forall$ (*old,* ϕ) $\in$ *memoryMap*,
    Delete *old*
  $\forall$ (ϕ*, new*) $\in$ *memoryMap*,
    Do nothing
  $\forall$ (*old, new*) $\in$ *memoryMap*,
    $\forall$ *field* $\in$ fields of pointer unswizzled type of *old*,
      Update *old.field* = *new.field*
      Delete *new*

deletes the new object. If the new object does not map to any old object, it retains the new object because it would have been created by the API. If the old object does not map to any new object, it deletes the old object because it would have been deleted by the API. An example of client address space is shown in Figure 4(b).

In Figure 4(b)., on the client, new objects pointed by *newRoot* need to be plugged in place of objects pointed by *oldRoot*. An example with addresses (0x10 – 0x60) is shown. Algorithm 1 constructs the memory map. Algorithm 2 typecasts the objects and copies their fields to the old objects. The plugged group of objects is shown at the bottom.

We generate Java functions statically for *computeMap(.)* (Algorithm 1) and *updateOldObjects(.)* (Algorithm 2) and call them after every API call in the Java code. In the Java implementation, we save both the data and the type in the objects so that depending on the runtime type of each object,



(a) Problem of plugging the objects returned from the server to the client. **(b)** Plugging back using Algorithms 1 and 2.

Fig. 4. Problem of plugging back objects returned by the server into the client address space is shown in (a). Solution is shown in (b).

Dataset	Description	ms	Classes	Methods	Fields	APIs	Pointers
DayTrader	Trading App	8	111	952	530	331	159
PBW	Online plant store	6	36	424	276	96	83
Acme-Air	Airline App	4	38	196	117	25	0
Petclinic	Vet-Clinic	4	37	138	57	7	0
Mayocat	E-Commerce	7	667	3042	1276	142	0

Fig. 5. Monolith applications studied for presence of pointers in data passing via APIs between microservices. Columns denote number of microservices, classes, methods, fields, APIs, and APIs with pointers, respectively.

reachable objects can be computed in *computeMap(.)* and its fields can be accessed and updated in *updateOldObjects(.)*.

IV. Experiments and Empirical Measurements

We study how prevalent is pointer communication while converting monolith to microservices.

A. Benchmarks

We study five publicly-available web-based monolith java applications, viz. Daytrader [6], Plantsbywebsphere [11], Acme-Air [12], Petclinic [13] and Mayocat [14]. These applications have been used in industry for converting monolith to microservices [1], [4].

B. Methodology

For each API, we check if the caller object, any parameter object, or return object or their field is referenced by any other object. We detect this using type analysis [7].

C. Empirical Measurements

Figure 5 records number of microservices (ms) recommended by clustering tool [1], number of classes, methods, fields, and APIs in the five monolith applications. Under the column labeled "Pointers", the table records the number of APIs that need communication of pointers. Larger the number of such APIs, more grave is the need of solving this problem. The table shows that several APIs pass objects containing pointers in DayTrader and PBW (Plantsbywebsphere). These objects require special handling (i) during serialization i.e., when objects are passed between client and server and (ii) during plugging back i.e. after the objects are returned from server to client.

V. Related Work

Microservices-based application follows distributed architecture, exposes different modules as services. Based on the load, services instances are scaled. Villamizar et al. presented a case study where they developed an enterprise application using both monolithic and microservice architecture and showed the benefits of microservices [15]. Broadly there are two ways to handle communication between services, synchronous and asynchronous message passing [16], [17]. Several protocols exist to support the communication. Remote Procedure Call used to be a popular choice that allows remote execution of a function in a different context. Even though it started with XML datatype, they extended to support other data formats like JSON, Protobuf, Thrift etc. Soon SOAP which is a purely XML-formatted, highly standardized web communication protocol became popular. It introduced Web Service Description Language (WSDL) where the endpoints are defined. REST based web services became popular and got used for information exchange. This is because it is comparatively lightweight and heavily tied with HTTP protocol [18]. Tihomirovs and Grabis present a performance review of REST and SOAP based web services where REST is shown to have

better performance [19]. Though REST based web services can be used with XML, binary objects, REST over JSON is the most popular implementation choice for developing microservices. Since microservices advocate functional independence, stateless messaging between microservices is preferred irrespective of the synchronous/asynchronous communication type. But this is difficult to achieve while translating monolith to microservices since the transferred data may container pointers. Our techniques of pointer swizzling and building a memory map are applicable for data transfer via any mode of communication.

Pointer swizzling [9], [10] is an old concept. It is used in object-databases. When an in-memory object is serialized and persisted, the pointers in it are converted to unique IDs. When the object is deserialized and loaded back to memory from the persistent store, the IDs are *swizzled* back to normal pointers. However, it has not been used to solve the problem highlighted in this paper. We use the database analogous to the communication channel between the microservices.

VI. Conclusions and Future Work

Several work exist that propose how to break a monolith application into microservices. However, no solution exists to enable communication of objects containing pointers between microservices. Unlike a monolith application, memory address space is not shared between microservices. Therefore, sharing of objects containing pointers cannot be done using their memory addresses. The presence of aliases makes the problem non-trivial.

In this work, we highlight two requirements: (i) accurate transfer of objects containing pointers between client and server address space, and (ii) plugging i.e., copying objects from server back to the client address space to update the state. JSON cannot be used for accurate transfer of objects because serialization into JSON loses alias information. Static alias analysis is imprecise and cannot be used to solve the problem of plugging back. We assign a unique global ID to each object. We apply pointer swizzling to solve the first problem. We propose creation of memory maps to solve the second problem.

In the future, we wish to study the performance impact due to proposed algorithms and how often non-trivial aliasing occurs on the client and server side.

References

[1] U. Desai, S. Bandyopadhyay, and S. Tamilselvam, "Graph neural network to dilute outliers for refactoring monolith application," in *Proceedings of 35th AAAI Conference on Artificial Intelligence (AAAI'21)*, 2021.

[2] S. Agarwal, R. Sinha, G. Sridhara, P. Das, U. Desai, S. Tamilselvam, A. Singhee, and H. Nakamuro, "Monolith to microservice candidates using business functionality inference," in *2021 IEEE International Conference on Web Services (ICWS)*. IEEE, 2021, pp. 758–763.

[3] A. Mathai, S. Bandyopadhyay, U. Desai, and S. Tamilselvam, "Monolith to microservices: Representing application software through heterogeneous gnn," *arXiv preprint arXiv:2112.01317*, 2021.

[4] A. K. Kalia, J. Xiao, C. Lin, S. Sinha, J. Rofrano, M. Vukovic, and D. Banerjee, "Mono2micro: an ai-based toolchain for evolving monolithic enterprise applications to a microservice architecture," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1606–1610.

[5] L. Nunes, N. Santos, and A. R. Silva, "From a monolith to a microservices architecture: An approach based on transactional contexts," in *European Conference on Software Architecture*. Springer, 2019, pp. 37–52.

[6] "Java ee7: Daytrader7 sample," https://github.com/WASdev/ sample.daytrader7.

[7] V. Kanvar and U. P. Khedker, "Heap abstractions for static analysis," *ACM Comput. Surv.*, vol. 49, no. 2, jun 2016. [Online]. Available: https://doi.org/10.1145/2931098

[8] V. Kanvar and U. Khedker, "What's in a name? going beyond allocation site names in heap analysis," *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, pp. 92–103, 2017.

[9] A. Kemper and D. Kossmann, "Adaptable pointer swizzling strategies in object bases: Design, realization, and quantitative analysis," *The VLDB Journal*, vol. 4, no. 3, p. 519–567, jul 1995.

[10] S. J. White and D. Dewitt, "Pointer swizzling techniques for object-oriented database systems," Ph.D. dissertation, 1994, aAI9434146.

[11] "Plants by websphere," https://github.com/WASdev/ sample.plantsbywebsphere.

[12] "Acme-air," https://github.com/acmeair/acmeair.

[13] "Spring petclinic sample application," https://github.com/spring-projects/spring-petclinic.

[14] "Mayocat shop," https://github.com/jvelo/mayocat-shop.

[15] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *2015 10th Computing Colombian Conference (10CCC)*. IEEE, 2015, pp. 583–590.

[16] P. Johansson, "Effcient communication with microservices," 2017.

[17] C. Richardson., "Building microservices: Inter-process communication in a microservices architecture." https://www.nginx.com/blog/building-microservices-inter-process-communication/, 2014.

[18] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[19] J. Tihomirovs and J. Grabis, "Comparison of soap and rest based web services using software evaluation metrics," *Information technology and management science*, vol. 19, no. 1, pp. 92–97, 2016.