Migrating the Communication Protocol of Client-Server Applications

Gabriel Darbord, Benoît Verhaeghe, Anne Etien, Nicolas Anquetil, Mustapha Derras

Department: Head
Editor: Name, xxxx@email

Migrating the Communication Protocol of Client-Server Applications

G. Darbord
Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

B. Verhaeghe
Berger-Levrault, France

A. Etien
Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

N. Anquetil
Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

M. Derras
Berger-Levrault, France

***Abstract*—As part of a collaboration with Berger-Levrault, an international IT company, we are working on the migration of client-server applications. To escape legacy technologies and to evolve towards a "software as a service" model, the company decided to migrate the client side of its applications to Angular 14 and the server side to Spring Boot. In this paper, we focus on the migration of client-server communication from RMI and GWT-RPC to the REST architectural style. We identify issues associated with such a migration and propose a tool-based approach to address them. The migration involves (1) identifying existing services and exchanged data structures; (2) migrating the services; (3) migrating the data structures on the new client side; and (4) in some cases, reducing the amount of exchanged data to address performance issues. We experimented with our approach on four of the company's applications currently using RMI or GWT-RPC.**

BERGER-LEVRAULT is an international software publisher providing management solutions for citizens, families, elected officials, healthcare providers, educators, and schools. Its applications rely on various technologies, some of which are very old and/or no longer maintained [1], [2]. The company has been developing software systems using Java technologies for more than 25 years, and with Google Web Toolkit (GWT) since its

 Published by the IEEE Computer Society

release. As part of a modernization effort to escape outdated legacy technologies, the company wants to migrate its applications to newer technologies. We are participating in a project to migrate client-server applications (Swing or GWT) to Angular (which uses TypeScript) for the client part, and Spring Boot for the server part. In this paper, we consider four industrial applications, with sizes between 300 and 2 700 KLOC, and counting between 500 and 4 400 services. The legacy client-server Swing application uses Java's Remote Method Invocation (RMI) for communication between the Java Swing client and the Java server. The legacy GWT applications use the GWT implementation of Remote Procedure Call (GWT-RPC) for communication between the Java GWT client and the Java server. For the new client-server communication, the company chose to develop an HTTP API using JavaScript Object Notation (JSON) to exchange data because these technologies are widely accepted de facto standard and is easy to implement in both Angular and Spring Boot. An HTTP API corresponds to a low level of REST maturity[1]. Aiming for a higher level would have required deeper changes and made the migration more difficult.

In all of these projects, the server will remain largely the same with only minor adjustments to handle the HTTP requests. The client part of the applications will be migrated to Angular using the approach proposed in [2] and will not be discussed here.

In this paper, we focus on the automatic migration of client-server communication from the legacy RMI and GWT-RPC to the planned HTTP API. In order to design a communication migration tool, we had to address the following four issues: (i) identify all the services and Data Exchange Classes; (ii) adapt how clients invoke server-side services; (iii) rewrite Data Exchange Classes in a different language; and (iv) in some cases, we had to minimize the amount of exchanged data to solve introduced performance issues. Data Exchange Classes correspond to the type of data structures that are transferred between the clients and the server, and Data Exchange Objects correspond to their instances. They can be thought of as Data Transfer Objects[2], but in our case some Data Exchange Objects contain behavior: methods that are more complex than simple accessors. In addition, for the remainder of this article, the term "calling convention" will be used to refer to the manner clients invoke server-side services.

We experimented with our approach on four applications from Berger-Levrault. We report here some results of the migrations. This is a work in progress, and the migrated applications are still undergoing validation by their respective development teams. Throughout the process, we are assisted by static analysis tools, including a custom linter that helps ensure the quality and consistency of the code to be migrated.

This experience report is structured as follows: first, we present the migration context and define the vocabulary used; then, we describe the issues encountered in this project; we continue by describing our solution; and finally, we report some data from running the tool on four industrial applications.

Migration Context

A migration approach to completely rewrite a monolithic GWT application into two separate projects, a JavaScript client and a Java server, communicating via a RESTful HTTP API is proposed in [3]. This approach allows to eliminate the technical debt in the old application, and create a client-server architecture with well-designed services. Berger-Levrault rejected this solution because it is costly and carries all the usual risks of software development projects. Instead, the company opted for an automated migration that might result in a less optimal application, but would deliver results faster and at a lower cost. In this context, understanding the client-server communication is essential for a successful migration. This section introduces the factors to consider.

Client-Server Communication refers to the way both sides of a distributed application interact and exchange data. To migrate client-server communication, we must consider the client's Web *Pages*, the *Services* provided by the server,

[1] https://martinfowler.com/articles/richardsonMaturityModel.html

[2] https://martinfowler.com/eaaCatalog/dataTransferObject.html

and the *Data Exchange Objects* that are exchanged between them.

The *Pages* contain the representation of the Graphical User Interface (GUI) and the behavioral code that is executed when the user interacts with it. Their migration is beyond the scope of this paper (see for example [2]). These pages display data obtained from the server's *services*.

The *Services* are server-side constructs that must also be described in the client to ensure interoperability. This enables the connection between the two parts of the applications. For example, in a human resources application, an "employee service" could be used to query employees. Technologies such as RMI and GWT-RPC in Java use proxies on the client to transparently manipulate objects created remotely on the server. When invoked, a client "service" (actually a service descriptor) creates a request to the corresponding server service and waits for the response. When it receives that response, it passes it back to the original caller (*i.e.* a page or another client service descriptor).

The application client and server exchange data through what we call *Data Exchange Objects*. A service response typically contains one or more instances of a Data Exchange Class, such as a list of employees associated with a particular request.

Migration Issues

To automatically migrate client-server communication from RMI and GWT-RPC to an HTTP API, the following issues must be addressed:

- All services must be automatically identified, as well as their calling and receiving locations in the source code;
- Data Exchange Classes, used to transfer data to and from services, must also be identified and their representation converted from Java or TypeScript to JSON (serialization/deserialization);
- The service calling convention must be adapted: with RMI and GWT-RPC, the client invokes a Java method directly, whereas with an HTTP API in Spring Boot, the client sends an HTTP request to the server, which is handled by a controller;
- In addition, we needed to minimize the amount of exchanged data (payload) in some cases.

Identifying the Services

To migrate the client-server communications, one must identify all calls to services in the client and all the definitions of those services in the server. We will see in a next section that this step is not difficult in the cases of RMI or GWT-RPC because they follow specific source code conventions (class annotation or class inheritance).

Identifying the Data Exchange Classes

The exchanged data must also be considered. First, one needs to identify what data is being exchanged, *i.e.*, what classes define Data Exchange Objects. Then, one must make sure that it can be serialized and deserialized correctly. RMI and GWT-RPC technologies rely on a proprietary binary serialization format that is handled transparently by the frameworks themselves. On the other hand, REST is typically associated with a generic and standard serialization format such as JSON, and data (de)serialization must be handled explicitly by the developer. This introduces some difficulties, which we explore in the following paragraphs.

A Data Exchange Class corresponds to a *Type* in Java, TypeScript, or any other language. The JSON format, which has its own MIME type of "application/json", contains only primitive types, arrays, and dictionaries. JSON primitive types are string, number, and boolean; JSON arrays can contain any number of "JSON types"; and JSON dictionaries are key-value collections where the keys are strings and the values are "JSON types". There is no structural description in JSON for runtime data. Objects are transferred as field-name/field-value dictionaries. This is a problem because dictionaries and objects that are serialized to JSON have exactly the same format.

Therefore, the serializer and the deserializer must agree on the structure of the transferred data beforehand. This is a known issue of "data format" described in the introduction to integration styles in [4]. Even then, there are still difficulties in the case of transferring polymorphic objects. For example, to deserialize a collection of `AbstractEmployee`, a deserializer would need to distinguish instances of `FullTimeEmployee` from instances

of `PartTimeEmployee`.

There is also the case of object references. If a field of a serialized object contains another object, the latter must be serialized recursively. If the second object contains a reference back to the first, this creates a known issue with serializing objects with circular references.

A final problem is that RMI and GWT-RPC ignore field visibility. When an object is sent from the server to the client, the frameworks are able to serialize all of the object fields, even the private ones. On the other hand, some modern JSON libraries, such as Jackson, only (de)serialize fields with public accessors.

Adapting the Calling Convention

The calling conventions for a service are different between RMI/GWT-RPC and an HTTP API.

RMI and RPC invoke a service by executing a remote class method. On the other hand, modern RESTful solutions invoke a service using an HTTP request comprising a resource locator (the URL) and an HTTP verb (POST, GET, PUT, DELETE, *etc.*[3]). Therefore, client-side invocations must be adapted to the conventions of an HTTP API.

Minimize Amount of Exchanged Data

After the initial experiments, we encountered some performance issues. While evaluating the performance of a migrated application, we noticed that the exchange of some messages became noticeably slower (from near instantaneous, to tens of seconds). This often happened on pages displaying a table of objects, where the full objects are transferred (as described by the Embedded Entity pattern in [5, p.314]) while only a few fields are actually displayed. Profiling showed that this was due to deserialization in the client for multi-megabyte payloads. This is a known problem with JavaScript JSON processing libraries, as noted in previous work on Web front-end migration [6]. Because RMI and GWT-RPC do not have this problem, the application developers did not focus on making smaller payloads.

[3]https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

Migration Process

This paper proposes a process for migrating client-server communication that addresses all of the above issues. We describe here our solutions for (i) identifying the services and Data Exchange Classes; (ii) adapting the calling conventions of services; (iii) migrating Data Exchange Classes; and (iv) pruning Data Exchange Objects for performance issues.

Services And Data Exchange Classes Identification

The services are identified using coding conventions, by static analysis of the applications. In the case of RMI, the framework requires service classes to implement the `Remote` interface, and their methods to be able to throw a `RemoteException` to be exposed. For GWT-RPC, service classes are required to implement the `RemoteService` interface, and all the public methods are exposed.

Once a method implementing a service is identified, its return and parameter types can be identified as Data Exchange Classes.

Adapt Calling Conventions

In the legacy applications, services are implemented by methods in specific classes (as previously mentioned). For ease of implementation, we decided to keep the same convention for the migrated applications. Thus, each class in a legacy implementation is migrated as a class in its new implementation, and each method in RMI or GWT-RPC is migrated as an HTTP API endpoint. To simplify the migration, we also use only the HTTP verb: POST. This results in operating at REST maturity level one.

On the server side, we automatically generated wrapper services which make use of Spring Boot, where they are called *controllers*. The generated controller for a `methodX` will accept HTTP requests and handle JSON deserialization, then delegate the execution of the service to the old `methodX`. This is illustrated in Figure 1 and Listing 1.

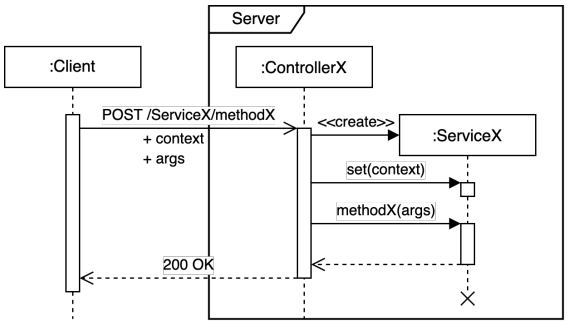

Figure 1: Wrapping of legacy RMI service into new Spring controller

```
1  @RestController
2  @PostMapping(path = "/ServiceX",
3    consumes = "application/json",
4    produces = "application/json")
5  public class ServiceXController {
6
7    @PostMapping("/methodX")
8    public ResultType methodX(
9      @RequestBody Map<String, Object> args)
10       throws RemoteException {
11     // get arguments from JSON payload
12     final ArgType1 arg1 = args.get("arg1");
13     final ArgType2 arg2 = args.get("arg2");
14     // get other arguments...
15     final User _user = args.get("_user");
16
17     final ServiceX service = new ServiceX();
18     service.setUser(_user);
19
20     return service.methodX(arg1, arg2, ...);
21   }
22 }
```

Listing 1: Automatically generated controller with a `methodX` API endpoint. The strings "arg1", "arg2", etc. represent the argument names of the old service's method.

Note that the `RemoteException` on line 10 is a remnant of the RMI implementation that required services to be able to throw this exception. The exception is there because we avoided modifying the original services, but it will never be thrown because RMI is no longer used. Another way to handle it would be to add a try block with an empty catch statement. If we had not decided to keep as much of the original code as possible, we could have refactored the original service to remove the reference to the RMI interface and not declare the throwing of the RMI exception.

In the legacy applications, services have a small context: an object that represents the authenticated user of the application. Our solution is to use the Context Representation pattern [5, p.96] and embed this small context in each request as a specific argument (line 15) of the service invocation. This context is then set in the `ServiceX` instance (line 18) before the method is called.

Data Exchange Class Migration

We discussed earlier that there are several problems associated with using JSON to transfer data: no structure description, polymorphic objects, circular references. We need to use a JSON library that handles these problems for us. But since the language is not the same for both communicating parts, Java on the server side and TypeScript on the client side, we need compatible libraries so that exchanged data is understood in the same way by both. We chose the Jackson library embedded in Spring Boot for the server side, and the Jackson-js library, an implementation of the Jackson library in JavaScript, for Angular.

As described previously, there is also a difference between serialization mechanisms based on attributes (ignoring their visibility) or based on public getters/setters. The Jackson library accesses attributes through public accessor methods. Again, because RMI and GWT-RPC give more freedom, application developers did not strictly implement all the accessors. We decided that this was a violation of good programming rules and programmed it into our (custom made) linter so that developers should fix all violations before the migration occurs.

As mentioned above, the Data Exchange Classes of the original applications sometimes contained behavioral code. While it is not complex to automatically migrate the Data Exchange Class hierarchy and fields from Java to TypeScript, these methods had to be ignored in the process due to the difficulty of transpilation. The development teams will need to reimplement them manually, and again we use our linter to identify all occurrences of such methods.

Pruning Data Exchange Objects

The last problem we had to deal with was the pruning of Data Exchange Objects. This is necessary when the server sends too much data to the client, resulting in a noticeable wait while the client deserializes it.

To solve this problem and reduce the payload size, we use a solution inspired by GraphQL[4] and follow the Wish List pattern [5, p.335]. The idea is to reduce the size of the Data Exchange Objects returned by the server, by not serializing data that is ignored by the client. In practice, when the client performs a request to the server, it adds, as a parameter, the list of fields of interest in the returned objects. Note that this list is not shown in Listing 1 (line 14) for the sake of clarity. Then, on line 20, before returning it to the client part, the result of the method call is passed to a special function along with the list of fields of interest. This function discards the uninteresting fields so that they do not clutter the JSON data. Again, for the sake of clarity, this function call is not shown in the listing.

GraphQL itself is a comprehensive technology that would have replaced Spring controllers, and required substantial changes to the applications' source code. Therefore, we chose the small *Jackson-AntPathFilter*[5] project, which provides the same pruning functionality but makes fewer assumptions about the host code.

The applications we worked on are all "standard management applications" (see below). As such, a large part of the UI displays tables of data. The cases where pruning was necessary were for these tables, when the service returns a list of objects of which only a few fields are actually displayed. For each displayed table, we identify the associated service invocation (which collects the data to display) and we extract the fields that the table actually requires.

To do this, we used static analysis of the code, which was facilitated by the fact that the developers followed code style guidelines.

First, we need to associate a displayed table with the service that provides its data:

- The table widget is identified because it is a class that inherits from `AbstractTableWidget`;
- We look for the creation of instances of this class and the use of the `setModel(new ATableModel())` method, which associates a model with the widget;
- Table model classes inherit from an `AbstractTableModel` superclass that provides a `loadData` method. We look for the call to this method in `ATableModel` since it usually takes as parameter the invocation of a service (which will return the required data).

This tells us which service returns the data that will be displayed in the table. We must now identify which fields are actually displayed so that we can prune the result of the service. To do this, we look at the constructor of the table widget, which contains calls to the `addColumn` method (see Listing 2). Each parameter of `addColumn` is an attribute of interest.

```
1 class FolderTableWidget
2   extends AbstractTableWidget {
3
4   public FolderTableWidget() {
5     addColumn(Folder.class, "date");
6     addColumn(Folder.class, "city.cityRef");
7   }
8 }
```

Listing 2: Initialization of a Table

This solution is not completely reliable, as it may sometimes be difficult to track down the service invocation from the loadData call. When we are not able to follow a complex data flow from the call to `loadData`, we insert a "TODO" comment in the generated code next to this call. The application developers will have to go back to the migrated client, to resolve all these comments by manually tracking down the service invocation and inserting the list of fields of interest.

Evaluation

We tested our solution on four real industrial applications. While we manually verified that the migrated applications do work as intended, they have not yet been accepted by their respective development teams.

Table 1: Four applications on which we tested our approach

	KLOC	Service classes	Service methods	Data Ex. Classes	Legacy comm.
app1	1 090	323	2 991	6 705	RMI
app2	2 735	448	3 355	4 611	GWT-RPC
app3	1 125	645	4 474	5 099	GWT-RPC
app4	322	63	534	479	GWT-RPC

Table 1 gives some data about the subject applications. They are management applications

[4] https://graphql.org/
[5] https://github.com/Antibrumm/jackson-antpathfilter

(financial, customers, human resources). Three of them are large, with hundreds of service classes (in the server part), and thousands of service methods (actual services). Prior to the migration, three of the applications used GWT-RPC and one used RMI.

To evaluate the usefulness of Data Exchange Object pruning in terms of execution time and memory consumption, we analyzed two different web pages. The first contains a seven-column table widget. It displays three rows after the service invocation. The second one contains an eight-column table widget. It displays nine rows after the service invocation. For each web page, we evaluate three aspects: the size (in kilobytes) of the message returned by the server, the time (in milliseconds) the server spent executing the request and pruning the result, and the "user time" (in milliseconds) from before the call to `loadData` to after it. This is the time the user waits for the empty displayed page to load the data. It includes the call to the service and the time to deserialize the Data Exchange Object.

Table 2: Time and memory consumption with and without object pruning (values averaged over several calls)

		Payload size (*Kb*)	Server time (*ms*)	User time (*ms*)
Case 1	no pruning	64	18	1 301
	w/ pruning	39	24	822
	Improvement	40%	-25%	37%
Case 2	no pruning	4 249	204	12 126
	w/ pruning	83	92	1 401
	Improvement	98%	55%	89%

Table 2 shows the results of this experiment. In the first case, the payload size is reduced by 40%, going from 64 Kb to 39 Kb. The server takes more time due to data pruning (24 ms instead of 18 ms), but the difference is negligible from a human perspective. The user time decreases from 1 301 ms to 822 ms (about 0.5 seconds), which is a small difference for the end user.

In the second case, the improvement in object size is huge (98%), going from 4 249 Kb to 83 Kb. The server time is also reduced, going from 204 ms to 92 ms (55%), probably because there is so much less data to serialize. The user time decreases from 12 126 ms to 1 401 (89%). This last case is a concrete example of a service for which the end user would have to wait more than 10 seconds, which drops to 1.5 seconds after pruning (still noticeable, but more acceptable).

The overall inferior performance of RESTful HTTP compared to RPC technologies is causing some companies to migrate back to RPC [7]. If performance becomes a significant issue in validating migrations, a more efficient JSON-compliant solution could be used at the expense of readability [8].

In summary, we have succeeded in designing a semi-automated tool to migrate services from legacy RMI/GWT-RPC to a modern Spring HTTP API. Using Data Exchange Object pruning, we also addressed the performance issue of (de)serialization.

Conclusion

In this paper, we expose a concrete problem of client-server communication migration. We presented an approach to migrate the client-server communication of applications and applied it to applications of Berger-Levrault.

Client-server migration is a complex and time-consuming process that requires careful planning and execution. To successfully migrate client-server communications, it is necessary to identify the technologies and protocols used by the existing system, as well as any potential roadblocks or compatibility issues. It is also important to have a clear understanding of the data exchanged between the client and server, as well as any dependencies or interactions between different components of the system.

We propose to use a semi-automated tool to identify and update code that is specific to the legacy system, while leaving more general code intact. In this context, standardizing the source code according to coding conventions can make the migration process smoother and more efficient. This reduces the manual effort of discovering and mapping source widgets to their target counterparts, and it improves the maintainability of the code.

Finally, it is important to thoroughly test the migrated system to ensure that it works correctly and meets the needs of end users.

We provide links at https://github.com/badetitou/Casino to several importers and generators that can help migrate applications,

and provide details about our approach.

REFERENCES

1. S. Bragagnolo, N. Anquetil, S. Ducasse, S. Abderrahmane, and M. Derras, “Analysing Microsoft Access projects: Building a model in a partially observable domain,” in *International Conference on Software and Systems Reuse (ICSR’20)*, ser. LNCS, no. 12541, Dec. 2020.
2. B. Verhaeghe, A. Shatnawi, A. Seriai, A. Etien, N. Anquetil, M. Derras, and S. Ducasse, “From GWT to Angular: An experiment report on migrating a legacy web application,” *IEEE Software*, 2021.
3. C. Zirkelbach, A. Krause, and W. Hasselbring, “On the modernization of ExplorViz towards a microservice architecture,” in *Combined Proceedings of the Workshops of the German Software Engineering Conference 2018*. CEUR Workshop Proceedings, 2018.
4. G. Hohpe and B. WOOLF, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, ser. The Addison-Wesley Signature Series. Prentice Hall, 2004. [Online]. Available: http://books.google.com.au/books?id=dH9zp14-1KYC
5. O. Zimmermann, M. Stocker, D. Lübke, U. Zdun, and C. Pautasso, *Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges*, ser. Addison-Wesley Signature Series (Vernon). Addison-Wesley Professional, 2022.
6. B. Verhaeghe, A. Shatnawi, A. Seriai, A. Etien, N. Anquetil, M. Derras, and S. Ducasse, “A hybrid architecture for the incremental migration of a web front-end,” in *Proceedings of the 17th International Conference on Software Technologies - ICSOFT*, INSTICC. SciTePress, 2022, pp. 101–110.
7. Y. Lee and Y. Liu, “Using refactoring to migrate REST applications to gRPC,” in *Proceedings of the 2022 ACM Southeast Conference*, ser. ACM SE ’22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 219–223. [Online]. Available: https://doi.org/10.1145/3476883.3520220
8. J. C. Viotti and M. Kinderkhedia, “Benchmarking JSON BinPack,” *ArXiv*, vol. abs/2211.12799, 2022.

Gabriel Darbord is a Ph.D. student at the RMoD Team of Inria Lille - Nord Europe, France. Contact him at gabriel.darbord@inria.fr.

Benoît Verhaeghe is a research engineer at Berger-Levrault. Contact him at benoit.verhaeghe@berger-levrault.com.

Anne Etien is full Professor at the University of Lille, France. Contact her at anne.etien@inria.fr.

Nicolas Anquetil is assistant Professor at the University of Lille, France. Contact him at nicolas.anquetil@inria.fr.

Mustapha Derras is research director at Berger-Levrault. Contact him at mustapha.derras@berger-levrault.com.