future internet

MDPI

Article

Automatically Injecting Robustness Statements into Distributed Applications

Daniele Marletta, Alessandro Midolo and Emiliano Tramontana *

Dipartimento di Matematica e Informatica, University of Catania, 95125 Catania, Italy; daniele.marletta@phd.unict.it (D.M.); alessandro.midolo@unict.it (A.M.)
* Correspondence: tramontana@dmi.unict.it; Tel.: +39-095-7383008

Abstract: When developing a distributed application, several issues need to be handled, and software components should include some mechanisms to make their execution resilient when network faults, delays, or tampering occur. For example, synchronous calls represent a too-tight connection between a client requesting a service and the service itself, whereby potential network delays or temporary server overloads would keep the client side hanging, exposing it to a domino effect. The proposed approach assists developers in dealing with such issues by providing an automatic tool that enhances a distributed application using simple blocking calls and makes it robust in the face of adverse events. The proposed devised solution consists in automatically identifying the parts of the application that connect to remote services using simple synchronous calls and substituting them with a generated customized snippet of code that handles potential network delays or faults. To accurately perform the proposed transformation, the devised tool finds application code statements that are data-dependent on the results of the original synchronous calls. Then, for the dependent statements, a solution involving guarding code, proper synchronization, and timeouts is injected. We experimented with the analysis and transformation of several applications and report a meaningful example, together with the analysis of the results achieved.

Keywords: resilience; refactoring; static code analysis; RMI; network faults; parallelism

Citation: Marletta, D.; Midolo, A.; Tramontana, E. Automatically Injecting Robustness Statements into Distributed Applications. *Future Internet* **2024**, *16*, 416. https://doi.org/10.3390/fi16110416

Academic Editors: Jerry Chou and Wu-Chun Chung

Received: 28 September 2024
Revised: 30 October 2024
Accepted: 8 November 2024
Published: 10 November 2024

1. Introduction

Distributed applications are crucial in many commercial, scientific, medical, and engineering fields; however, their development presents many more challenges than centralized ones, mainly due to network delays, faults, potential attacks, performance, etc. [1]. Distributed applications often use a mechanism such as Java Remote Method Invocation (RMI), or similar ones in other programming languages, i.e., a blocking call, to simplify the code that calls a service on a remote server.

In a distributed system, components spread across the network can be affected by communication delays, which deteriorates performance [2,3]. Additionally, the potential inadequate management of exceptions and failures during client–server communication can severely compromise the resilience and robustness of distributed applications, creating critical failure points [4]. The blocking of remote method calls cannot handle delays or disruption on the distributed service on its own. Moreover, some solutions based on simple RMI expose further vulnerabilities [5–7].

Sure, advanced solutions have been suggested to better face possible communication delays and service unavailability, such as the design of pattern timeouts, circuit breakers, etc. [8]. Moreover, it could be that the developer knows that some service execution is long enough to require an asynchronous call; then, e.g., messaging solutions are used [9], or ad hoc integration is required due to its characteristics, e.g., with a blockchain [10]. Nevertheless, blocking calls and RMIs are widely used and recent studies show the usefulness of RMIs, e.g., in cloud systems like Amazon Web Services in other cloud computing environments, in enterprise solutions, in legacy systems, etc. [11–13].

As an example, in a cloud-based e-commerce platform, services like inventory management and payment processing use RMIs. Then, due to network delays, responses can be slow, causing performance issues and order confirmation delays. Moreover, poor exception handling may result in critical errors, affecting the way users can interact with the distributed system. Overall, the persistent use of blocking calls, like RMIs, underscores the need for improved resilience in distributed applications, as these calls struggle with the management of delays and failure scenarios, ultimately compromising system robustness.

To address these limitations, this paper aims to provide an approach and a support tool that analyzes distributed applications to find the use of simple remote calls (such as RMIs) and substitute them with a more advanced fragment of code that, while handling the remote call, additionally takes into account delays, faults, service disruption, etc. Simple remote calls have been found by tracking all calls to methods that were in classes implementing the Java Remote interface. While the initial remote call is a blocking one, we introduce an asynchronous call to manage network delays and faults. This has the benefit of making the application parallel, hence making it better equipped to be reactive. Of course, the necessary code for synchronization and guarding will be inserted to have the correct execution of data dependent statements. Distributed systems can significantly take advantage of parallel execution to optimize their performance [14].

While libraries for supporting the development of distributed and parallel applications exist for many languages, including both Java and C++ [15,16], they typically require developers to manually identify which remote calls should be parallel and use appropriate APIs to initialize, launch, and synchronize other execution threads. Our approach, instead, while avoiding blocking calls, generates parallel versions of the source code for such calls, freeing developers of such issues, while also providing them with the source code of the accurate transformation. Relevant related work has aimed to automatically parallelize executable code, achieving significant performance enhancements [17–19]. However, these approaches conceal the parallelized code from developers and were not aimed at distributed applications.

Our approach, along with the corresponding tool, employs data dependence analysis (following Bernstein's conditions [20]) to determine the appropriate code transformations and synchronization necessary for a reliable distributed and parallel execution. First, the source code is analyzed using the Javaparser library to detect blocking remote calls. Next, data dependence analysis is performed to find instructions depending on the result of the remote calls. Specifically, two statements have a data flow dependence when the output set (i.e., the set of variables written) of the first instruction overlaps with the input set (i.e., the set of variables read) of the second instruction [20–22]. Finally, the program's dependencies are updated according to the concurrency and failure handling library used, ensuring the integrity of the application. All of the above steps are executed automatically in a given source code. We performed several experiments to evaluate the benefits of our approach and analysis. An application was selected to apply the correspondent refactoring, and its performances were assessed using specific benchmarks, demonstrating an effective reduction in execution time.

The main contribution of this paper is a comprehensive approach that automatically transforms a Java source code with blocking remote calls into a parallel and fail-safe code. By leveraging advanced static analysis techniques and data dependence analysis based on Bernstein's conditions, our method identifies and refactors specific fragments of code. This transformation not only improves performance but also enhances the resilience of distributed systems by incorporating robust concurrency and failure-handling mechanisms.

The remainder of this paper is structured as follows. Section 2 reviews the state-of-the-art approaches and compares existing approaches with our proposed method. Section 3 introduces our general approach and presents a high-level architecture for analyzing and transforming the code. Section 4 describes the methodology for automatically transforming blocking calls into asynchronous calls and checking data dependencies among instructions. Section 5 presents the analysis and transformation performed on a sample application,

along with the execution results. Section 6 discusses the obtained results and the limitations of our approach. Finally, conclusions are drawn in Section 7.

2. Related Works

The Java RMI mechanism is frequently used in distributed systems to enable communication between client components and a server side. The state of the art encompasses numerous approaches that analyze the Java RMI supporting library, focusing on aspects such as security, cryptography, asynchronous computing, and cloud computing.

A tool that encrypts and decrypts data, subsequently transmitting the data via the internet using Socket and RMI technology, was developed and proposed in [5]. The proposed approach employed the Cipher Block Chaining (CBC) mode of Advanced Encryption Standard (AES) algorithms to encrypt data after clients partition the data from a large file. The encrypted data are then sent to a daemon for parallel map-reduce processing. Moreover, RMIs are used in a cloud environment, and to facilitate development, language translators were used to convert design concepts to Java RMIs and provide the way for deployment on Amazon Web Services (AWS) Elastic Cloud Computing (EC2) [12].

A comprehensive assessment of RMI vulnerabilities using the Metasploit framework was presented in [6]. Encryption algorithms were investigated to ensure the security of transmitted data or objects using RMI [7]. Specifically, the authors applied AES and Data Encryption Standard (DES) algorithms, comparing their efficacy in conjunction with the RMI APIs. Such approaches focused on various applications of the Java RMI library; however, none of them aimed at enhancing the robustness and efficiency of RMIs. To the best of our knowledge, our proposed approach is novel in that it automatically transforms RMIs used in an application into code, handling the calls in a dedicated thread (asynchronous) and adding actions for recovery in case of potential faults or relevant delays.

Concurrent computing has gained significant popularity with the advent of multicore hardware. Numerous studies have proposed the design of automatic tools to efficiently refactor sequential code into its parallel equivalent. Asynchronous programming is prevalent in Android applications due to UI access and I/O operations [23]. Lin et al. [24] proposed a tool to automatically detect long-running operations and refactor them into asynchronous ones. Additionally, Ozkan et al. [25] described a tool for identifying the improper use of asynchronous constructs and making necessary changes. Unlike our approach, these methods are apt for analyzing parallel code to find defects. JavaScript ecosystems also provide synchronous and asynchronous calls for handling various I/O operations. Gokhale et al. [26] presented a refactoring approach to assist developers in transforming synchronous operations into asynchronous ones. Arteca et al. [27] proposed an approach to reorder asynchronous calls for earlier execution, yielding significant performance benefits. This method requires the input code to be parallel, with developers choosing the parts to parallelize. In another approach, Schäfer et al. [28] proposed lock refactoring to automatically convert built-in monitor locks in Java's synchronized blocks to the locks provided by the java.util.concurrent.locks library, such as ReentrantLock and ReadWriteLock. Various analyses ensured that the transformations preserved application behavior and enhanced performance. Several approaches focus on refactoring for loops to make them suitable for parallel executions [29,30]. In contrast to the previous approaches, our proposal provides (i) automation in the identification of the method calls that can be executed in parallel (which are RMI calls), and (ii) the automatic injection of statements that make the execution of identified calls both parallel and robust. This frees developers from manually identifying the suitable parts of code and implementing more robust code.

Zhang et al. [31] presented an automated approach to convert synchronized locks into StampedLock. Later, Zhang et al. [32] proposed a prototype for automatically converting coarse-grained locks into fine-grained locks to reduce lock contention and improve performance and scalability. These approaches focused on modernizing and optimizing existing parallel code, letting the developer determine which code segments to parallelize. Our approach, on the contrary, takes as input a sequential software system and identifies

where parallel constructs could be used for improving performance and scalability while preserving correctness.

Kaminsky and Haidl [15,16] introduced two libraries for Java and C++, respectively. These APIs support multicore parallel programming, cluster parallel programming, GPU acceleration, and big data parallel programming. Developers are asked to manually integrate these features to convert sequential code to parallel. However, identifying suitable code fragments for parallel execution and selecting the appropriate concurrent APIs pose significant challenges for developers [33,34]. Our proposal addresses these issues automatically by selecting necessary instructions by means of a detailed analysis and generates refactored parallel versions using appropriate libraries.

In [34,35], the authors presented two approaches for applying atomic refactoring and collection refactoring [36] to refactor synchronized statements. They proposed several transformations: converting Int to AtomicInteger, Long to AtomicLong, HashMap to ConcurrentHashMap, WeakHashMap to ConcurrentWeakHashMap, and HashSet to ConcurrentHashSet. The focus was on modernizing existing parallel code using the libraries introduced since Java 5. The effectiveness of these modifications was tested for correctness. Additionally, Dig et al. [34] suggested refactoring sequential recursive algorithms to parallel versions using ForkJoinTask, demonstrating performance benefits through the evaluations of popular recursive algorithms. Conversely, our approach transforms sequential code to parallel by adding new threads, unlike other methods that merely update class types. It is also less invasive than ForkJoinTask, requiring only the use of CompletableFuture and get() calls. Additionally, it applies to all method calls meeting certain preconditions, whereas other methods are limited to specific cases like recursive algorithms and Java synchronized blocks.

Several approaches focus on optimizing compiled code to achieve parallelism [17–19]. These methods analyze executable code to identify coarse-grained tasks, such as the iterations of large loops, and execute them in parallel. Although these approaches demonstrate considerable performance gains, the optimization process remains opaque to the developer.

3. Proposed Approach for Improving Remote Calls

3.1. Overview

When executing remote calls, the called services run on a different host, which can lead to delays due to relatively slower network communication, necessary security checks, data retrieval processes, and the potential workload of the remote server. To address these issues and improve performance and robustness, one effective strategy is to execute remote operations in a newly dedicated thread. Through offloading these operations to a separate thread, it becomes possible to handle them asynchronously, hence allowing the caller to execute other tasks, hence avoiding a blocking call, and then improving overall robustness, efficiency and responsiveness.

To facilitate the development of a distributed application that includes such an advanced handling of remote calls, the overall strategy of the proposed approach entails analyzing the source code of a distributed Java application to identify remote calls, and then transforming them into new code fragments that preserve the application's intended behavior while enhancing performance, such as through asynchronous execution and improved resilience to network faults.

Implementing asynchronous calls requires careful consideration of concurrency issues, such as race conditions and data consistency. Proper synchronization mechanisms must be employed to ensure that concurrent threads do not interfere with each other, leading to unpredictable behavior or data corruption. Conducting a data dependency analysis can help mitigate these risks by examining the instructions following the calls that are changed into asynchronous and inserting the necessary synchronization measures.

Figure 1 illustrates the three main phases of the proposed approach: (i) remote call identification, which parses the Java code to locate all instances of remote calls (including, e.g., RMIs); (ii) data dependency analysis and code transformation, which replaces remote

calls with new fragments of code that enhance the application's functionalities; (iii) project document update, which involves verifying and updating the application dependencies.

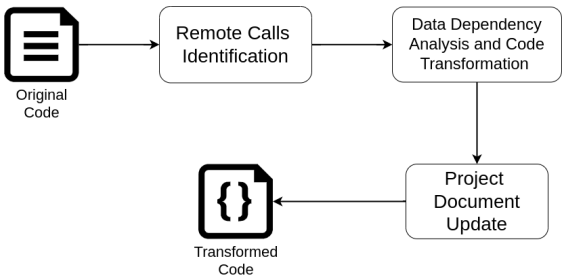

Figure 1. Overview of the three main phases of the proposed approach: in the first phase, the source code is parsed to locate remote calls (Section 3.2). Next, the identified calls are analyzed to ensure data consistency, and the code is transformed to support parallel execution and fault tolerance (Section 3.3). The final phase involves updating the application's dependencies (Section 3.4).

Firstly, to identify remote calls in a Java source file, the code needs to be parsed. We used the JavaParser library (https://javaparser.org, accessed on 18 September 2024), which simplifies this task. A custom Visitor class was developed to examine the code and find classes implementing the Remote interface (defined in the standard Java library under java.rmi) and the track method calls on such classes. Secondly, the found method calls are then further analyzed to address potential synchronization, allowing for data consistency, and refactored to ensure asynchronous behavior and fault tolerance. Finally, application dependencies are updated using Apache Maven. The pom.xml file is modified to include all necessary dependencies for the transformed code, ensuring that the application's functionality is maintained.

The final output of this process is represented by the transformed source code that, upon compilation and execution, offers enhanced functionalities. The following sections detail the three phases of the proposed approach, respectively.

3.2. Remote Call Identification

To identify remote calls within Java source files, the code is parsed and analyzed using the JavaParser library. This library constructs an Abstract Syntax Tree (AST) [37], a hierarchical structure that represents the code, with the root node representing the entire file and all other elements as child nodes. This approach provides a comprehensive overview of the source code's structure.

To facilitate code inspection, the VoidVisitorAdapter class from JavaParser was employed, which enables the definition of a custom Visitor class designed to search for specific properties within the source code. The visit() method implemented in the Visitor class accepts two parameters: the type of object being examined (e.g., interface declaration, method declaration) and a container for storing the retrieved data.

The remote call identification phase comprises three steps: (i) extracting all classes implementing an extension of the Remote interface; (ii) analyzing the variable declarations and finding all instances of the types identified in the previous step; (iii) examining method calls to identify all invocations that correspond to remote calls. The Remote interface acts as a marker for interfaces whose methods can be invoked from a different JVM. Any object identified as remote must directly or indirectly implement this interface. Consequently, only methods declared within a remote interface (i.e., one that extends java.rmi.Remote) can be invoked remotely.

For the first step that identifies classes implementing at least one remote interface, a Javaparser visitor VoidVisitorAdapter<HashSet<ClassOrInterfaceDeclaration>> is used. This visitor traverses the source code AST, and retrieves the list of implemented interfaces by invoking the getImplementedTypes() method on the AST nodes. Subsequently, all interfaces extending Remote are filtered and added to the HashSet.

In the second step, a visitor VoidVisitorAdapter<HashSet<VariableDeclaration>> traverses the AST and inspects all variable declaration statements. If the declared type is found within the HashSet from the previous step, the corresponding instance is tracked by adding it to the visitor's HashSet.

In the final step, all method invocations on tracked instances are assessed by evaluating the MethodCallExpr expressions associated with such instances. For each identified MethodCallExpr, a series of operations are performed to determine the type associated with the method call. This aims at tracing the call's origin and identifying the invoked method. If the invocation corresponds to a method of a remote interface, it is then classified as a remote method call. Once identified, it serves as the input for the data dependency analysis and code transformation phase, as described in the following Section 3.3.

3.3. Data Dependency Analysis and Code Transformation

Transforming calls into asynchronous execution requires careful analysis to ensure program correctness. A key aspect of this process is managing shared data across threads, which is essential for preventing concurrency issues such as race conditions and data inconsistency. Data dependency analysis reveals the relationships among instructions that are subsequent to the asynchronous call. This analysis uncovers potential conflicts that may arise when multiple threads simultaneously access or modify shared variables. Hence, appropriate synchronization mechanisms are introduced to ensure safe execution and maintain data integrity.

According to the results of the data dependency analysis, the automatic transformation of remote calls can be safely performed, and to improve application robustness, the modification relied on the Failsafe library (https://failsafe.dev, accessed on 18 September 2024). Failsafe provides a comprehensive set of flexible, composable policies specifically designed for failure detection, handling, and recovery. Moreover, these resilience policies seamlessly integrate with CompletableFuture, a feature in Java's standard library (java.util.concurrent), to enable asynchronous operations. By combining Failsafe's resilience mechanisms with the asynchronous capabilities of CompletableFuture, applications can achieve not only enhanced fault tolerance but also improved overall performance. The analysis and the transformation are further discussed in Section 4, where a comprehensive examination is provided, including detailed analysis and significant examples.

3.4. Project Document Update

In the final phase of the proposed approach, application dependencies are updated. Numerous tools and systems exist to manage dependencies, automate tasks such as fetching correct versions, resolving conflicts, and ensure that all necessary dependencies are available.

For Java language, Apache Maven (https://maven.apache.org/, accessed on 18 September 2024) has emerged as a prominent software project management tool, specifically designed to facilitate the management of project dependencies and streamline the build process. Maven relies on a Project Object Model (POM), an XML file that contains project information and configuration details. The <dependencies> section of the POM file lists all external libraries required by the project.

For our goal, the pom.xml inspection and update were based on the MavenXpp3Reader class, which, like JavaParser, provides means for parsing and building a corresponding tree model. Once the tree representation was generated, the group, artifact, and version nodes required for Failsafe were inserted. This was achieved by creating a new node of type Dependency. After modifying the tree, the updated pom.xml file was written, ensuring that the application had all the dependencies needed to execute the transformed code.

4. Dependency Analysis and Code Transformation

4.1. Data Dependency Analysis

When aiming at transforming the code to introduce in it asynchronous calls, threads have be appropriately guarded to ensure safety and prevent issues due to the access of shared data. Our proposed transformation approach performs a thorough inspection of RMIs and the following instructions to identify potential dependencies. The analysis relies on Bernstein's Conditions [20], a set of criteria used to determine the independence of instructions in concurrent processing. Bernstein's Conditions involve examining two sets of variables for each instruction: the input set and the output set. The input set comprises all variables that the instruction reads, while the output set includes all variables that the instruction writes. The not empty intersection of the output set for the RMI with the input set of the subsequent instructions indicates that the subsequent instructions are dependent on the output of the RMI [38].

Algorithm 1 outlines the steps for performing the data dependence analysis. Firstly, it retrieves the input set (variables read by the statement) and the output set (variables written by the statement) for the identified RMI. Secondly, for each statement following the RMI statement, it computes the intersection between these sets. If the intersection contains at least one element, the current instruction is identified as data-dependent, and synchronization has to be inserted at this point to ensure correctness. Otherwise, if no intersection was found, the next instruction will be analyzed. If no data-dependent statements were identified, then the last instruction of the method will be the point of code where synchronization will be handled.

Algorithm 1 The operations performed to identify the data-dependent instructions.

function DATADEPENDENCEANALYSIS(m, rmi)
  $inputSetS_1 \leftarrow getInputSet(m, rmi)$
  $outputSetS_1 \leftarrow getOutputSet(m, rmi)$
  for $in \leftarrow m.getFollowingInstructions(rmi)$ **do**
    $inputSetS_2 \leftarrow getInputSet(m, in)$
    $outputSetS_2 \leftarrow getOutputSet(m, in)$
    if $checkIntersections(inputSetS_1, outputSetS_1, inputSetS_2, outputSetS_2)$ **then**
      return in
    end if
  end for
  $lastInstruction \leftarrow getLastInstruction(m)$
  return $lastInstruction$
end function

Listing 1 provides an example of code for which data dependence analysis was performed. Firstly, line 5 was identified as an RMI suitable for the asynchronous execution transformation, as the method service.getAuthors() is invoked on a type declared as a subtype of Remote. From line 6 onward, the analysis seeks to detect data-dependent statements, identifying such a statement in line 8 due to the use of the variable authors for subsequent Java Stream operations. This data-dependent statement marks the synchronization point, i.e., the original thread has to wait for the completion of the forked thread.

Additionally, line 6 contains another RMI, a call to another method of a subtype of Remote. The analysis determined that the variable book is in the output set of the instruction. Then, a subsequent data-dependent statement was found in line 8 (book appears in line 10; however, it is in the same statement starting in line 6). Given that no statements are present between the remote call (line 6) and the dependent statement (line 8), then no parallelism can be achieved for the two said lines.

Listing 1. An example of data-dependent instructions within a method declaration. The method calls at line 5 and 6 are remote method calls; line 8 is data-dependent on line 5 and 6.

```
1  public class Client {
2  private static DataInterface service = new ServiceProxy();
3
4  public static Optional<Author> extractMostReviewedAuthor() {
5  HashMap<String, Author> authors = service.getAuthors();
6  Book book = service.getMostReviewedBook();
7
8  return authors.values().stream()
9  .filter(auth -> auth.getBooks().stream()
10 .anyMatch(b -> b.getTitle().equals(book.get().getTitle())))
11 .findFirst();
12 }
13 }
```

4.2. Code Transformation

After performing the data dependency analysis described above, the found RMIs can be automatically refactored to ensure failure handling and asynchronous execution. The transformation employs the Failsafe library, which has been available since Java 8. Failsafe is instrumental in managing failure cases with various patterns and policies, as it offers multiple policies handling specific exceptions encountered during network communication. The appropriate response to exceptions depends on the selected policy.

To enhance the client application, we implemented a Wait and Retry policy with exponential backoff. This approach entails that when an exception occurs during communication, the RMI call is retried after an exponentially increasing interval. As the number of consecutive exceptions grows, the waiting period before each retry increases. This strategy significantly enhances the robustness of the application by allowing it to overcome intermittent communication issues.

Moreover, Failsafe enables the asynchronous execution of RMI by integrating the CompletableFuture library. CompletableFuture represents a future result of an asynchronous computation and provides a rich set of methods for creating, chaining, and combining asynchronous tasks. This integration can result in significant performance enhancements in Java applications [39]. By leveraging CompletableFuture within Failsafe, we notably improve the robustness and efficiency of applications, ensuring the proper handling of RMIs.

The code shown in Listing 1 has been automatically transformed into the one shown in Listing 2. For the latter code, lines 7 and 8 are the refactored version of the RMI (in line 5 of the former code) using the CompletableFuture and the FailSafe libraries. The data dependency analysis has identified the instruction at line 13 as data-dependent; hence, the synchronization was inserted just before, at line 11, using the f.get() method, which returns the result obtained by the previous call to getAuthors() method once it is available, hence blocking execution when needed. The instructions were encapsulated within a try/catch statement (lines 6 and 18) to enhance the robustness and guarantee that a missed answer of the server will be handled properly.

Similarly to the previous remote call at line 5 in the original code, line 6 of the original code has been refactored using the Failsafe library and is in line 9 of the transformed code. This refactoring aims at achieving robustness but without introducing asynchronous execution, given the subsequent data dependency. The integration of the Failsafe library enhances the RMI's resilience against failures, particularly network instability or transient errors, thereby maintaining operational integrity.

Listing 2. An example of the transformed code for the origin code shown in Listing 1. The getAuthors() RMI (lines 7 and 8) now uses CompletableFuture to achieve asynchronous execution and FailSafe to achieve robustness, whereas the getMostReviewedBook() RMI only uses FailSafe to achieve robustness. The synchronization instruction is inserted at line 11.

```
1  public class Client {
2  private static DataInterface service = new ServiceProxy();
3  ....
4
5  public static Optional<Author> extractMostReviewedAuthorParallel(){
6  try {
7  CompletableFuture<HashMap<String, Author>> f =
8  Failsafe.with(retryPolicy).getAsync(() -> service.getAuthors());
9  Book book = Failsafe.with(retryPolicy)
10 .get(() -> service.getMostReviewedBook());
11 HashMap <String, Author> authors = f.get();
12
13 return authors.values().stream()
14 .filter(auth -> auth.getBooks().stream()
15 .anyMatch(b -> b.getTitle().equals(book.get().getTitle())))
16 .findFirst();
17 }
18 catch(Exception e){
19 e.printStackTrace();
20 return Optional.empty();
21 }
22 }
23 }
```

The Failsafe class accepts a RetryPolicy object as input, which determines the policy to be followed for retry operations. The RetryPolicy is shown in Listing 3. Each method on the RetryPolicy builder ensures that a specific property is set: (i) handle(Exception.class) specifies that the policy should handle all exceptions of type Exception; (ii) withBackoff(1, 30, ChronoUnit.SECONDS) establishes the backoff strategy for retries, beginning with a 1 s wait time and increasing exponentially up to 30 s; (iii) withMaxRetries(20) limits the number of retry attempts to twenty; and finally, (iv) onRetriesExceeded(e -> failsSafeCallback(e)) defines an action to be executed when the maximum number of retries is reached, where a callback function failsSafeCallback(e) is automatically created to ensure the safe handling of the failure mechanism. The developer can customize the body of the callback function to handle exceptions in a manner that aligns with the underlying software requirements. Finally, build() finalizes and constructs the RetryPolicy object.

In our transformed code, a RetryPolicy object is inserted as an attribute into the class containing the analyzed method. This approach ensures that the specified retry policy is applied to all RMIs within the class, thereby avoiding code repetitions and promoting consistency in handling retry logic.

Listing 3. The configuration of the retry policy used for all RMIs transformed using the FailSafe library.

```
1 RetryPolicy<Object> retryPolicy = RetryPolicy.builder()
2 .handle(Exception.class)
3 .withBackoff(1, 30, ChronoUnit.SECONDS)
4 .withMaxRetries(20)
5 .onRetriesExceeded(e -> failSafeCallback(e))
6 .build();
```

5. Results

To evaluate the gains of our proposed code transformations, we performed experiments on a sample distributed application designed to extract data from the Amazon Books

Reviews dataset [40], enabling the inference of characteristics related to books, authors, and reviews. The dataset consists of around 3 million book reviews authored by a large user base, and covers a diverse corpus of 212,404 distinct books.

To ensure a manageable execution time during testing for transformation correctness and performance evaluation, a subset of around 200,000 reviews were selected. The application's source code is publicly accessible via a dedicated repository [41]. Figures 2 and 3 show the UML class diagrams outlining the application's components.

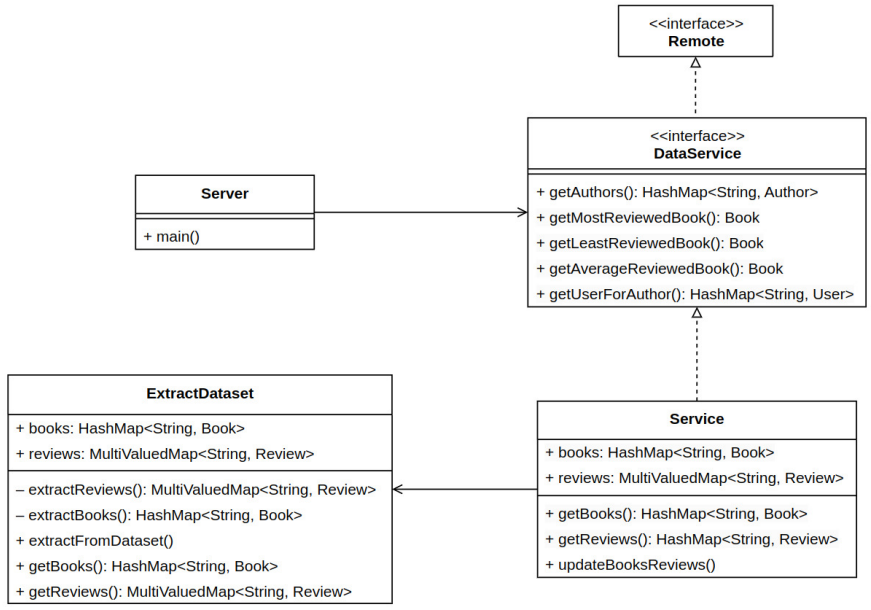

Figure 2. The UML class diagram of the server-side architecture.

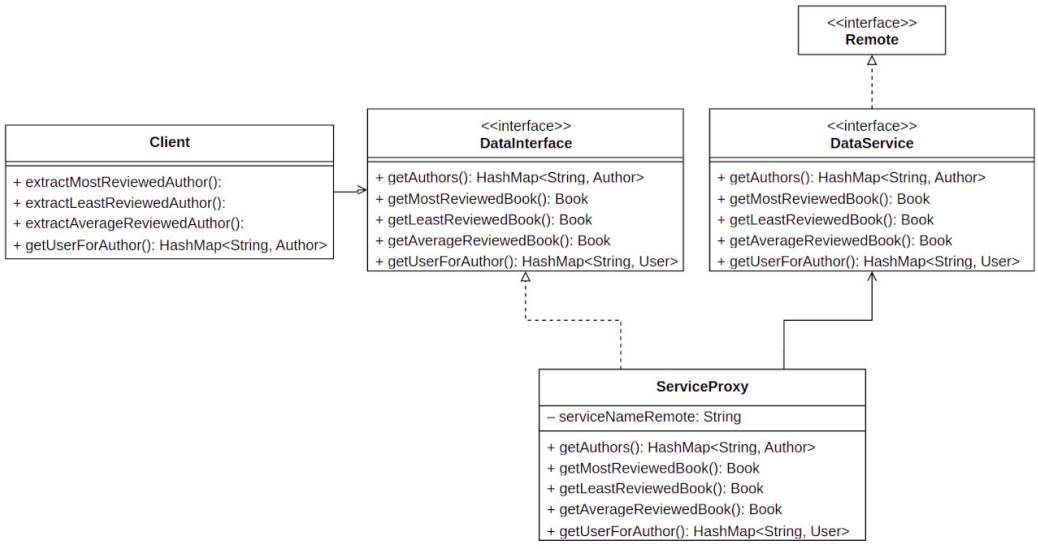

Figure 3. The UML class diagram of the client-side architecture.

The DataService remote interface defines five methods: getAuthors(), getMostReviewedBook(), getLeastReviewedBook(), getAverageReviewedBook(), and getAverageReviewedBook(). The Server class creates and configures an instance of Service, which implements the DataService interface and uses the methods of the ExtractDataset class to retrieve information from the book dataset.

On the client side, the ServiceProxy class implements DataService and manages network communication, performing RMIs on the methods exposed by Service. The Client

class creates a ServiceProxy object to connect to the service and receive the results of remote method executions.

An analysis of the Client class identified four methods suitable for transformation, each involving two distinct RMI calls to extract some data from the dataset. As seen previously, Listing 1 shows a method of the origin Client class that was automatically converted into the robust and parallel version shown in Listing 2.

To evaluate the performance of the refactored version, we used the Java Microbenchmark Harness (JMH), a standardized tool for conducting rigorous performance tests. Our setup included warming up the execution environment to ensure reliable performance metrics: each benchmark consisted of five warm-up iterations followed by ten normal iterations. JMH tests are particularly suited for assessing performance improvements as they isolate the evaluated subject from external influences.

We performed four benchmarks for each refactored method. Each benchmark table shown below follows a consistent structure: the "ith-run" column indicates the iteration number; "sequential time (ms)", and "parallel time (ms)" display the execution time in milliseconds per iteration for the two versions; and "speed-up" denotes the performance gain achieved by the parallel version (computed as $((runtime_{old} - runtime_{new})/runtime_{old}) \times 100$). This allows for a detailed comparison of performance improvements achieved in the refactored version, validating its effectiveness in enhancing application robustness and efficiency under controlled testing conditions.

Tables 1, 2, 3 and 4 show, respectively, the resulting benchmarks for the following methods: extractAverageReviewedAuthor(), extractLeastReviewedAuthor(), extractMostReviewedAuthor(), and getUserForAuthor().

Table 1. Execution times for ten runs of the extractAverageReviewedAuthor() method.

ith Run	Sequential Time (ms)	Parallel Time (ms)	Speed-Up
1	6540	6321	3.35
2	6045	6111	−1.09
3	6418	5723	10.83
4	5869	5930	−1.04
5	6084	5027	17.37
6	7034	5603	20.34
7	6118	4663	23.78
8	5877	5563	5.34
9	6186	4944	20.08
10	6785	4995	26.38
avg	6295	5488	12.82

Table 2. Execution times for ten runs of the extractLeastReviewedAuthor() method.

ith Run	Sequential Time (ms)	Parallel Time (ms)	Speed-Up
1	7821	6950	11.14
2	8311	5454	34.38
3	7138	7272	−1.88
4	7445	5642	24.22
5	7051	7512	−6.54
6	7958	7505	5.69
7	7525	5921	21.32
8	7777	6721	13.58
9	7578	7187	5.16
10	7615	6928	9.02
avg	7621	6709	11.97

Table 3. Execution times for ten runs of the extractMostReviewedAuthor() method.

ith Run	Sequential Time (ms)	Parallel Time (ms)	Speed-Up
1	6721	5732	14.72
2	5967	5023	15.82
3	5846	5503	5.87
4	6221	4429	28.81
5	5847	5343	8.62
6	6153	5830	5.25
7	6661	6003	9.88
8	5713	5943	−4.03
9	5821	4872	16.30
10	6188	5685	8.13
avg	6113	5436	11.08

Table 4. Execution times for ten runs of the getUserForAuthor() method.

ith Run	Sequential Time (ms)	Parallel Time (ms)	Speed-Up
1	12,231	7979	34.76
2	11,499	9518	17.19
3	11,524	9212	20.08
4	11,748	9149	22.14
5	11,580	7667	33.78
6	12,110	9634	20.43
7	11,956	10,232	14.40
8	11,683	9582	17.96
9	10,458	9222	11.82
10	10,417	7853	24.60
avg	11,520	9004	21.83

An additional benchmark was executed to assess the resilience of the distributed system both before and after our transformation. Table 5 presents the benchmark results for the getUserForAuthor() method, which was executed while simulating communication failures during the RMI calls. The "ith run" column indicates the iteration number, while the "delay (ms)" column displays the network delay in milliseconds occurring during the RMI call. The columns labeled "rmi-time (ms)" and "mod-time (ms)" represent the execution times for the original RMI version and the modified RMI version proposed by our approach, respectively. Additionally, the "# of retries" column shows the number of retries (additional calls after some faults was notified) performed in the modified version by the FailSafe library, according to the code shown in Listing 3.

Table 5. Execution times for ten runs of the getUserForAuthor() method.

ith Run	Delay (ms)	rmi-Time (ms)	mod-Time (ms)	# of Retry
1	50	10,485	7799	0
2	75	10,535	7850	0
3	150	10,656	7981	0
4	300	10,926	8255	0
5	600	11,426	8778	0
6	1200	12,126	9488	2
7	2400	-	10,883	3
8	4800	-	13,478	5
9	9600	-	18,493	10
10	19,200	-	38,523	20

6. Discussion

To evaluate the effectiveness of our approach, the transformed version was compiled and executed successfully. While the refactored version is robust against network delays and intermittent malfunctioning, it also performs better in terms of execution time. Across all four benchmarked methods, the average speed-up was 14.42%. The method getUserForAuthor() showed the highest improvement at 21.83%, while extractMostReviewedAuthor() exhibited the lowest at 11.08%. The performance of the parallel version and the resulting speed-up are primarily influenced by the workload balance between the parallel paths. Maximum gain occurs when both paths execute with minimal time discrepancy, enabling concurrent execution without idle time.

The benchmarks revealed varying speed-up results. The method getUserForAuthor() demonstrated high performance gain as the parallel threads exhibited similar execution times. Conversely, methods yielding 11.08%, 11.97%, and 12.82% speed-ups showed greater execution time discrepancies between parallel paths. Nonetheless, our transformations effectively enhanced performance across these benchmarks. Notably, getUserForAuthor() required longer execution times compared to other benchmarks due to its handling of larger datasets, necessitating greater computational effort, which in turn led to higher performance gains. The first three benchmarks (see Tables 1–3) exhibited some negative speed-ups, indicating that sequential execution achieved lower overall times. This may be attributed to factors such as thread scheduling, cache effects, and load balancing in parallel execution environments. In particular, the management of multiple threads can introduce synchronization delays, while frequent cache misses and the imbalanced distribution of tasks across threads can degrade performance, offsetting the benefits of parallelism.

When occasional network delays or failures occur, while the original version simply hangs or terminates abruptly, our transformed version of the application recovers from the faults, given that the network communication will be restored within the interval set in the proper RetryPolicy object.

Table 5 shows the significant impact of a communication delay. By default, if a Java RMI call fails, it does not automatically initiate a retry; rather, it throws a RemoteException (or a related exception depending on the issue), causing the operation to terminate unless properly handled. When the underlying socket timeout was set to 2 s, the RMI call would experience a crash each time the network communication delay exceeds 2 s. In contrast, our proposed transformation enhances the robustness and resilience by employing a retry policy, which make the running application recover from the timeout and remain operational despite increasing communication delays. As shown in Listing 3, this policy employs an exponential backoff mechanism with a maximum timeout of 30 s and allows up to 20 retries, thereby ensuring a robust and effective failure management.

Our approach was designed to enhance both the robustness of network communication and correct execution, while also aiming at achieving some performance gains. However, there are inherent limitations, particularly when relying solely on static code analysis.

Static analysis cannot reliably predict the specific method executed at runtime, especially in cases involving polymorphism. In these scenarios, we adopt a cautious approach for safety reasons then assume maximum data dependencies among the methods and potentially miss opportunities for parallel execution. Moreover, when code contains multiple references to objects of the same type, static analysis struggles to distinguish between instances, particularly when variables conditionally assign different references. This can lead to missed opportunities for parallel execution on distinct objects.

7. Conclusions

This paper presented a novel and effective methodology for the automatic identification and transformation of remote calls within Java source files, enhancing the overall functionality of applications. This enhancement includes enabling asynchronous execution and increasing resilience to network faults. Our methodology begins by processing sequential code to identify remote calls, followed by a comprehensive data dependency analysis to

ensure that the transformations preserve the intended behavior of the application and maintain execution correctness. Selected remote calls are then refactored using suitable libraries to optimize the code. The transformed code leverages the benefits of parallel execution, which significantly reduces the execution time. Additionally, the enhanced resilience to network faults ensures that the application can handle disruptions more gracefully, thereby improving overall reliability.

The effectiveness of our approach was validated through empirical testing on a distributed application that makes multiple remote calls to request and retrieve data from a real-world dataset. The results demonstrate a significant reduction in execution time while maintaining the application's correctness, showcasing the efficacy of our transformations. To the best of our knowledge, this approach marks the first attempt to automatically transform remote calls with the dual objective of adding support for asynchronous and fail-safe execution.

In conclusion, our approach offers a powerful tool for automatically refactoring and improving the performance and robustness of code in distributed applications.

Author Contributions: Conceptualization, E.T.; methodology, A.M. and E.T.; software, D.M., A.M. and E.T.; validation, D.M., A.M. and E.T.; formal analysis, D.M., A.M. and E.T.; resources, D.M. and A.M.; data curation, D.M. and A.M.; writing—original draft preparation, D.M. and A.M.; writing—review and editing, E.T.; visualization, D.M. and A.M.; supervision, E.T.; project administration, E.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: No data were created.

Acknowledgments: The authors acknowledge the support of the University of Catania PIACERI 2020/22 Project "TEAMS".

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Wollrath, A.; Riggs, R.; Waldo, J. A Distributed Object Model for the Java System. *Comput. Syst.* **1996**, *9*, 265–290.
2. Maassen, J.; Van Nieuwpoort, R.; Veldema, R.; Bal, H.; Kielmann, T.; Jacobs, C.; Hofman, R. Efficient Java RMI for parallel programming. *ACM Trans. Program. Lang. Syst.* **2001**, *23*, 747–775. [CrossRef]
3. Sharp, M.; Rountev, A. Static Analysis of Object References in RMI-Based Java Software. *IEEE Trans. Softw. Eng.* **2006**, *32*, 664–681. [CrossRef]
4. Bhamidipaty, A.; Lotlikar, R.; Banavar, G. RMI: A Framework for Modeling and Evaluating the Resiliency Maturity of IT Service Organizations. In Proceedings of the IEEE International Conference on Services Computing (SCC 2007), Salt Lake City, UT, USA, 9–13 July 2007; pp. 300–307. [CrossRef]
5. Duc Phung, T.; Van Nguyen, T.; Quy Tran, B. Design and Implementation a Secured and Distributed System using CBC, Socket, and RMI Technologies. In Proceedings of the International Conference on Software and Computer Applications (ICSCA), Kuala Lumpur, Malaysia, 23–26 February 2021; pp. 238–243. [CrossRef]
6. Joshi, A.P.; Kaur, N.; Kaur, A. Unveiling the Achilles' Heel: A Comprehensive Exploration of Vulnerabilities in RMI and Bindshell Communication. In Proceedings of the International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO), Noida, India, 14–15 March 2024; pp. 1–5. [CrossRef]
7. AL-Taie, R.R.K.; Abu-Alsaad, H.A. Application of Encryption Algorithms with RMI Protocol. In Proceedings of the International Arab Conference on Information Technology (ACIT), Zarqa, Jordan, 10–12 December 2020; pp. 1–6. [CrossRef]
8. Richardson, C. *Microservices Patterns: With Examples in Java*; Manning Publications Co.: Shelter Island, NY, USA, 2019.
9. Hohpe, G.; Woolf, B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*; Addison-Wesley: Boston, MA, USA, 2012.
10. Mandarino, V.; Pappalardo, G.; Tramontana, E. A Blockchain-Based Electronic Health Record (EHR) System for Edge Computing Enhancing Security and Cost Efficiency. *Computers* **2024**, *13*, 132. [CrossRef]
11. Basanta-Val, P.; Fernández-García, N.; Sánchez-Fernández, L. Predictable remote invocations for distributed stream processing. *Future Gener. Comput. Syst.* **2020**, *107*, 716–729. [CrossRef]
12. Godla, S.R.; Fikadu, G.; Adema, A. Socket Programming-Based RMI Application for Amazon Web Services in Distributed Cloud Computing. In *Innovative Data Communication Technologies and Application, Proceedings of the ICIDCA 2021, Coimbatore, India, 20–21 August 2021*; Lecture Notes on Data Engineering and Communications Technologies; Springer: Singapore, 2022; Volume 96, pp. 517–526.

13. Seemakhupt, K.; Stephens, B.E.; Khan, S.; Liu, S.; Wassel, H.; Yeganeh, S.H.; Snoeren, A.C.; Krishnamurthy, A.; Culler, D.E.; Levy, H.M. A Cloud-Scale Characterization of Remote Procedure Calls. In Proceedings of the 29th Symposium on Operating Systems Principles, 2023, SOSP '23, Koblenz, Germany, 23–26 October 2023; pp. 498–514. [CrossRef]
14. Wu, Z.; Sun, J.; Zhang, Y.; Wei, Z.; Chanussot, J. Recent Developments in Parallel and Distributed Computing for Remotely Sensed Big Data Processing. *Proc. IEEE* **2021**, *109*, 1282–1305. [CrossRef]
15. Kaminsky, A. Parallel Java Library. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA, 16–21 November 2014.
16. Haidl, M.; Gorlatch, S. PACXX: Towards a Unified Programming Model for Programming Accelerators Using C++14. In Proceedings of the LLVM Compiler Infrastructure in HPC (LLVM-HPC), New Orleans, LA, USA, 17 November 2014; pp. 1–11. [CrossRef]
17. Kimura, K.; Taguchi, G.; Kasahara, H. Accelerating Multicore Architecture Simulation Using Application Profile. In Proceedings of the IEEE International Symposium on Embedded Multicore/Many-Core Systems-on-Chip (MCSOC), Lyon, France, 21–23 September 2016; pp. 177–184. [CrossRef]
18. Kasahara, H.; Kimura, K.; Adhi, B.A.; Hosokawa, Y.; Kishimoto, Y.; Mase, M. Multicore Cache Coherence Control by a Parallelizing Compiler. In Proceedings of the IEEE Annual Computer Software and Applications Conference (COMPSAC), Turin, Italy, 4–8 July 2017; Volume 1, pp. 492–497. [CrossRef]
19. Felber, P.A. Semi-automatic Parallelization of Java Applications. In On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE, Proceedings of the OTM Confederated International Conferences CoopIS, DOA, and ODBASE 2003 Catania, Sicily, Italy, 3–7 November 2003; Springer: Berlin/Heidelberg, Germany, 2003; pp. 1369–1383. [CrossRef]
20. Bernstein, A.J. Analysis of Programs for Parallel Processing. *IEEE Trans. Electron. Comput.* **1966**, *EC-15*, 757–763. [CrossRef]
21. Maydan, D.E.; Hennessy, J.L.; Lam, M.S. Efficient and Exact Data Dependence Analysis. *ACM SIGPLAN Not.* **1991**, *26*, 1–14. [CrossRef]
22. Wolfe, M.; Banerjee, U. Data Dependence and Its Application to Parallel Processing. *Int. J. Parallel Program.* **1987**, *16*, 137–178. [CrossRef]
23. Dig, D. Refactoring for Asynchronous Execution on Mobile Devices. *IEEE Softw.* **2015**, *32*, 52–61. [CrossRef]
24. Lin, Y.; Radoi, C.; Dig, D. Retrofitting concurrency for android applications through refactoring. In Proceedings of the ACM International Symposium on Foundations of Software Engineering (FSE), Hong Kong, China, 16–21 November 2014; pp. 341–352. [CrossRef]
25. Ozkan, B.K.; Emmi, M.; Tasiran, S. Systematic asynchrony bug exploration for android apps. In Proceedings of the International Conference on Computer Aided Verification, San Francisco, CA, USA, 18–24 July 2015; Springer International Publishing: Berlin, Germany, 2015; pp. 455–461. [CrossRef]
26. Gokhale, S.; Turcotte, A.; Tip, F. Automatic Migration from Synchronous to Asynchronous JavaScript APIs. *Proc. ACM Program. Lang.* **2021**, *5*, 1–27. [CrossRef]
27. Arteca, E.; Tip, F.; Schäfer, M. Enabling Additional Parallelism in Asynchronous JavaScript Applications. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Aarhus, Denmark, 11–17 July 2021. [CrossRef]
28. Schäfer, M.; Sridharan, M.; Dolby, J.; Tip, F. Refactoring Java Programs for Flexible Locking. In Proceedings of the International Conference on Software Engineering (ICSE), Honolulu, HI, USA, 21–28 May 2011; pp. 71–80. [CrossRef]
29. Khatchadourian, R.; Tang, Y.; Bagherzadeh, M. Safe automated refactoring for intelligent parallelization of Java 8 streams. *Sci. Comput. Program.* **2020**, *195*, 102476. [CrossRef]
30. Midolo, A.; Tramontana, E. Refactoring Java Loops to Streams Automatically. In Proceedings of the 4th International Conference on Computer Science and Software Engineering, 2021, CSSE '21, Singapore, 22–24 October 2021; pp. 135–139. [CrossRef]
31. Zhang, Y.; Shao, S.; Liu, H.; Qiu, J.; Zhang, D.; Zhang, G. Refactoring Java Programs for Customizable Locks Based on Bytecode Transformation. *IEEE Access* **2019**, *7*, 66292–66303. [CrossRef]
32. Zhang, Y.; Shao, S.; Zhai, J.; Ma, S. FineLock: Automatically refactoring coarse-grained locks into fine-grained locks. In Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA), Los Angeles, CA, USA, 18–22 July 2020; pp. 565–568. [CrossRef]
33. Ahmed, S.; Bagherzadeh, M. What Do Concurrency Developers Ask about? A Large-Scale Study Using Stack Overflow. In Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Oulu, Finland, 11–12 October 2018. [CrossRef]
34. Dig, D.; Marrero, J.; Ernst, M.D. Refactoring Sequential Java Code for Concurrency via Concurrent Libraries. In Proceedings of the IEEE International Conference on Software Engineering (ICSE), Vancouver, BC, Canada, 16–24 May 2009; pp. 397–407. [CrossRef]
35. Ishizaki, K.; Daijavad, S.; Nakatani, T. Refactoring Java Programs Using Concurrent Libraries. In Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD), Toronto, ON, Canada, 17 July 2011; pp. 35–44. [CrossRef]
36. Dig, D.; Marrero, J.; Ernst, M.D. How Do Programs Become More Concurrent: A Story of Program Transformations. In Proceedings of the ACM International Workshop on Multicore Software Engineering (IWMSE), Honolulu, HI, USA, 21–28 May 2011; pp. 43–50. [CrossRef]
37. Smith, N.; Van Bruggen, D.; Tomassetti, F. *Javaparser: Visited*; Leanpub: Victoria, BC, Canada, 2017; Volume 10, pp. 29–40.

38. Midolo, A.; Tramontana, E. An API for Analysing and Classifying Data Dependence in View of Parallelism. In Proceedings of the Proocedings of the International Conference on Computer and Communications Management (ICCCM), Okayama, Japan, 29–31 July 2022; pp. 61–67. [CrossRef]
39. Midolo, A.; Tramontana, E. An Automatic Transformer from Sequential to Parallel Java Code. *Future Internet* **2023**, *15*, 306. [CrossRef]
40. Amazon Book Reviews Dataset. Available online: https://www.kaggle.com/datasets/mohamedbakhet/amazon-books-reviews (accessed on 15 July 2024).
41. RMI Book Reviews. Available online: https://github.com/AleMidolo/RMI_Books_Reviews (accessed on 15 July 2024).