Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Σχεδίαση και Υλοποίηση Κατανεμημένου Συστήματος για την Αυτόματη Αξιολόγηση Προγραμματιστικών Ασκήσεων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΝΙΚΟΛΑΟΣ ΤΣΙΑΜΗΤΡΟΣ

Επιβλέπων : Νικόλαος Παπασπύρου
Αναπ. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούνιος 2015

Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Σχεδίαση και Υλοποίηση Κατανεμημένου Συστήματος για την Αυτόματη Αξιολόγηση Προγραμματιστικών Ασκήσεων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΝΙΚΟΛΑΟΣ ΤΣΙΑΜΗΤΡΟΣ

Επιβλέπων : Νικόλαος Παπασπύρου
Αναπ. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 19η Ιουνίου 2015.

Νικόλαος Παπασπύρου Αναπ. Καθηγητής Ε.Μ.Π.	Κωνσταντίνος Κοντογιάννης Αναπ. Καθηγητής Ε.Μ.Π.	Δημήτριος Φωτάκης Επίκ. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούνιος 2015

Νικόλαος Τσιαμήτρος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Τα κατανεμημένα συστήματα έδωσαν τη δυνατότητα σχεδιασμού γρήγορων, αποκρίσιμων και ανθεκτικών στις αποτυχίες εφαρμογών. Τρέχοντας σε περισσότερα από ένα μηχανήματα κάθε στιγμή, μπορούν να χωρίσουν το συνολικό φορτίο σε κομμάτια και να τα μοιράσουν σε διαφορετικούς κόμβους, ελαχιστοποιώντας έτσι τον χρόνο που απαιτείται για να ολοκληρωθεί μια εργασία ή ένα σύνολο από εργασίες που διαφορετικά θα εκτελούνταν ακολουθιακά. Μια τέτοια εφαρμογή, αν σχεδιαστεί σωστά, μπορεί ιδανικά να εξυπηρετήσει τις αιτήσεις όλων των πελατών που δυνητικά μπορεί να έχει, χωρίς να ανησυχεί ποτέ για τον πραγματικό τους αριθμό ή για τον όγκο εργασίας που απαιτούν οι αιτήσεις τους. Η ανάγκη για μια παρόμοια κατανεμημένη εφαρμογή που μπορεί να κλιμακώνει, ώστε να αναλαμβάνει μεγάλα φορτία με αποτελεσματικό τρόπο, είναι το κίνητρο πίσω από τη συγκεκριμένη διπλωματική εργασία.

Αρχίσαμε δουλεύοντας με ένα υπάρχον σύστημα, το οποίο χρησιμοποιείται σε διάφορα μαθήματα του πολυτεχνείου και στον Πανελλήνιο Διαγωνισμό Πληροφορικής, για να λαμβάνει και να αξιολογεί προγράμματα, που υποβάλλονται ως λύσεις σε προκαθορισμένα προβλήματα. Στόχος μας ήταν να επεκτείνουμε το σύστημα για να το καταστήσουμε κλιμακώσιμο καθώς και να παρέχουμε τις ικανότητές του ως μια δημόσια διαθέσιμη υπηρεσία. Για να πετύχουμε αυτούς τους σκοπούς, σχεδιάσαμε μια κατανεμημένη αρχιτεκτονική, η οποία κατανέμει τις εισερχόμενες αιτήσεις για αξιολόγηση σε πολλαπλούς εργάτες αξιολόγησης και παρέχει μια κατάλληλη διεπαφή για να υπάρχει απομακρυσμένη πρόσβαση στις δυνατότητες αυτές.

Στη συνέχεια, υλοποιήσαμε την αρχιτεκτονική μας, στην οποία ενσωματώσαμε το αρχικό σύστημα, το οποίο δρα ως εργάτης αξιολόγησης (ή απλά εργάτης). Έπειτα, δοκιμάσαμε το σύστημα για να επαληθεύσουμε την αποτελεσματικότητά του και διαπιστώσαμε ότι η αναμενόμενη επιτάχυνση στη διαδικασία αξιολόγησης επιτυγχάνεται.

Λέξεις κλειδιά

Grader, σύστημα αξιολόγησης προγραμματιστικών ασκήσεων, Hellenico, κατανεμημένο σύστημα, ZeroMQ, RESTful API, Django REST framework, NFS, message broker, thread pool pattern.

Abstract

Distributed concurrent systems have enabled the design of fast, responsive, and fault-tolerant applications. Running on more than one single machine at any given instance, they can split the total workload and dispatch the resulting chunks to different nodes, thus minimizing the time needed to complete a task or a set of tasks that would otherwise be executed sequentially. Such an application, if designed properly, could ideally serve all its potential client requests without ever worrying about their actual number or the amount of work their requests demand. The need for a similar distributed application that can scale to successfully undertake big workloads is the motivation behind this particular thesis.

We started with an existing system, which is used in various university courses and in the Greek Computing Olympiad for high-school students, for receiving and evaluating programs, submitted as solutions to predefined problems. Our aim was to expand this system to make it scalable and provide its resources as a publicly available service. To accomplish these targets, we designed a distributed architecture, which distributes incoming evaluation requests to multiple evaluation workers and provides an appropriate interface to access those resources remotely.

Subsequently, we implemented our architecture to which we incorporated the original system, which acts as an evaluation worker (or simply worker). Then, we tested the system to verify its effectiveness, and we concluded that the expected speedup in the evaluation process is achieved.

Key words

grader, evaluation system, Hellenico, distributed system, ZeroMQ, RESTful API, Django REST framework, NFS, message broker, thread pool pattern

Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή Νικόλαο Παπασπύρου, που μου έδωσε την ευκαιρία να πραγματοποιήσω αυτήν τη διπλωματική εργασία και με βοήθησε καθοριστικά στην προσπάθεια ολοκλήρωσής της.

Ακόμη, θα ήθελα να ευχαριστήσω ιδιαίτερα τον Ιωάννη Χατζημίχο, που (μαζί με το Χρήστο Τζάμο) υλοποίησε το αρχικό σύστημα αυτόματης αξιολόγησης, πάνω στο οποίο στηρίχτηκε αυτή η διπλωματική εργασία, και ο οποίος με καθοδήγησε στη σχεδίαση και υλοποίηση του νέου συστήματος και μου προσέφερε ουσιαστική βοήθεια σε όλα τα στάδια της εργασίας.

Επίσης, οφείλω ένα ευχαριστώ στον Άγγελο Γιάντσιο για τη συμβολή του τόσο μέσω των συζητήσεων που είχα μαζί του όσο και μέσω της τεχνικής υποστήριξης που μου προσέφερε.

Τέλος, ευχαριστώ την οικογένεια και τους φίλους μου για την αμέριστη στήριξή τους σε όλη την διάρκεια της φοιτητικής μου διαδρομής.

Νικόλαος Τσιαμήτρος,

Αθήνα, 19η Ιουνίου 2015

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-4-15, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιούνιος 2015.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The purpose of this thesis is to design and implement a distributed system for receiving and evaluating programs, submitted as solutions to predefined problems. It is based on an existing automatic evaluation system, which is used for programming competitions and class assignments. We refer to this system as **evaluation system** or **grading system**.

A typical use case of the system, for example during a competition, consists of the following steps:

1. The competition administrators provide the specifications of the problems to be solved. For each problem, called **task**, these include information concerning the input and output filnames of the solutions, the time and memory limits and the test cases to be used. Each **test case** consists of an (input, expected output) pair. The **input** is an arbitrary byte sequence conforming to the format described in the task specifications and is provided as input to solutions. The **expected output** contains the value which solutions considered as "correct" are expected to produce.
2. The participants design and implement algorithmic solutions for each of the tasks and submit their source code to the evaluation system.
3. The system compiles the submitted solutions received, called **submissions** and runs the resulting executables against a set of the test cases available.
4. The participants receive feedback concerning the performance of their solutions, either in real-time or at the end of the competition.

The focus of the particular thesis is to expand the previous system, in order to achieve two goals:

- To provide the ability to simultaneously process multiple submitted solutions and increase the availability and resilience of the system.
- To create and expose a public interface, which will provide a unified way to access the grading resources, in order to offer grading as a service.

To achieve the first goal we designed and implemented a distributed system that uses multiple computer nodes to deploy instances of the automatic evaluation system, while it also exploits the parallelism capabilities inside each multi-core node, deploying multiple instances of the

system simultaneously. The system then takes up the task of receiving new submitted solutions from clients and passing them to one of the available evaluation systems running on the nodes. Using our architecture, the system can withstand failures and recover effectively from them.

To provide a grading service, we designed and implemented a REST API, which we then exposed publicly. This way, any evaluation system can benefit from the service, in order to provide the desired capabilities to its clients, without the need to implement a back-end, that would actually do the evaluation and grading.

1.1 Thesis motivation and background

The motivation behind this thesis emerged from the need to increase the performance and availability of the existing evaluation system, which does not implement any form of parallelism in the submission evaluation process. The system receives submitted solutions, compiles them appropriately, executes them against a given set of test cases in a sandboxed environment, and evaluates them based on the output they produce. As only a single grading node is used, it examines the incoming submissions sequentially, one at a time. Consequently, under situations of intensive work, such as a competition, a lab or an assignment deadline, this can result in big queues of submissions waiting to be processed, delaying feedback to the students and possibly making the system unresponsive.

Moreover, the existing, single-node system lacks availability. In case of node failure, it becomes inaccessible and cannot accept new submissions or evaluate the ones in queue, which is highly undesirable, especially in the case of a competition. We need to expand the system to withstand node failures and, when not possible, we must ensure that it will be able to fully recover its previous state, by eliminating the possibility of data loss.

It becomes evident, that the only way to improve the performance of the system and to make it reliable and highly available is to make it distributed, exploiting the parallelism capabilities of a multi-core, multi-node cluster environment.

Furthermore, creating a distributed evaluation system that can successfully handle big workloads, provides the chance to offer the resources for wider use, as a service, for other academic divisions or individuals. Therefore, it was necessary to design and expose a RESTful API to provide unified access to our resources, through standard HTTP requests.

Finally, the automatic evaluation system has been proven valuable, as it has been used for many years, having become an integral part of many classes offered at the National Technical University of Athens for automatically evaluating and grading student solutions to class assignments, and of the Panhellenic Competition in Informatics, which uses it for the needs of the competition. It is therefore important to provide a scalable and distributed, but above all reliable and fault-tolerant evaluation service, which is the main focus of this particular thesis.

1.2 Thesis structure

The thesis is organized as follows:

Chapter 2:
We provide the necessary theoretical background for the concepts and entities that are being discussed throughout the thesis.

Chapter 3:
We present the architecture of the existing evaluation system and describe its function.

Chapter 4:
This chapter provides a thorough explanation of our design proposal. We present our architecture and describe in detail its individual components and the way they provide our system with the desired properties.

Chapter 5:
We provide a comprehensive description of the way we implemented the various components of our system that we presented in the previous chapter.

Chapter 6:
We evaluate the performance of our design and implementation.

Chapter 7:
We provide some concluding remarks about our thesis and discuss some ideas for future work that will further expand and improve our system.

Chapter 2

Theoretical Background

2.1 Distributed Systems

A distributed system consists of multiple hardware or software components, located at different, networked nodes, which cooperate and communicate for a common purpose solely via message passing [2] [6]. This definition, and the properties presented below, become clear in Chapter 4, where we present the design of our distributed system.

2.1.1 Properties of Distributed Systems

The most significant properties of distributed systems are:

- *Concurrency of components*: In the network of nodes comprising the system, typically many programs run concurrently, possibly by different users, sharing system resources. The system structure is dynamic and can change by adding/removing resources to/from the network, while processing a distributed task.
- *No global clock*: Since the coordination of components is done by exchanging messages through a network, it is a challenging task for the computers to accurately synchronize their clocks, resulting in the absence of a single global notion of the correct time.
- *Independent component failures*: The independent nature of the distributed components means that the system can fail in many different ways. One or more components may crush or become inaccessible, due to network faults, leaving other parts of the system still running, often unaware of the situation.

2.1.2 Architectures

The architecture of a distributed system specifies the way its discrete components are structured, interrelated and interconnected. It usually falls under one of the following architectural models:

- *Client-Server*: In this model there are two distinct entities: the client and the server. The system components can be classified as either of them or sometimes even as both, based on their role. The server's role is to provide a service by receiving and responding

to requests from clients. The clients are the service requesters that communicate with the server to consume its products and they use the data they receive to perform tasks. Typically, there is a limited number of servers, often just one, and many clients connected to them. Consequently, the provision of service is centralized and the system's scaling capabilities limited. However, since this model provides a direct and relatively simple method to share resources and data, it remains the most widely used. This model is used by the architecture we design and present in Section 4.2, for the broker-worker interaction.

- *Peer-to-Peer*: This architecture describes systems, in which all components involved in a task have similar roles, dividing the workload among them. There is no distinction between client and server, but all components send and receive data and provide memory and processing power to the system, interacting as peers. As a consequence, as the number of components increases, so are the resources available to run the service. We should mention that we do not use this model in our architecture.

2.1.3 Communication Paradigms

Since the components of distributed systems are distinct, living on the same or separate networked computer nodes, they need to communicate in order to coordinate and exchange data. There are many ways through which this is achieved, but they usually follow one, or both, of the following basic paradigms:

- *Remote invocation*: This type of communication involves a bidirectional data exchange between two components of a distributed system, which results to the remote invocation of a procedure on one of the two components.

- *Indirect communication*: In this paradigm the communication between two components is indirect, which means that it does not take place directly between them, but it is accomplished through a third entity. As a result, the two communicating parties do not need to exist at the same time (temporal uncoupling) and the sender does not need to know the receiver (spatial uncoupling).

We will examine three particular examples that follow the aforementioned paradigms. These include the request-reply protocols and remote procedure calls, which follow the paradigm of remote invocation, and message queues, which provide a technique for indirect communication. These techniques are important to the implementation of the distributed system we intend to build.

Request-reply protocols

Request-reply protocols are a group of protocols following the remote invocation paradigm, used to support a Client-Server communication architecture. They usually start with a request from the client to the server, in the form of a message exchange, which results to the invocation of an operation on the server that processes the request. The results of this operation are sent back to the client as a message response, encoded as an array of bytes. The message exchange in such protocols is typically synchronous, in the sense that the client actively waits

for the response through an open connection to the server, but can also be implemented in an asynchronous fashion. As we present in section 4.2, an asynchronous type of request-reply protocol is used in the architecture we design for this thesis for the communication of the workers with the broker.

Remote procedure calls

Remote procedure call (RPC) is a technique employing remote invocation, that provides the means for a program running on a computer node to call a procedure located at a remote computer in the same way it calls a local one. This is done by providing an interface that hides the underlying operations taking place, such as message passing and encoding/decoding of parameters and results, necessary for the communication of a distributed system. This technique is very useful in client-server systems, where the servers support specific operations, which they make available to the clients through an API, and the clients call these operations as though they were local. From the client's view, the only visible part of the whole process is that they call a function and some time later the results are returned. In this thesis, we use this technique for the HTTP servers described in Section 4.2 to handle remote client requests, as well as for the broker-worker interaction.

Message queues

Message queues are a type of indirect communication that provides a point-to-point service through which components can exchange data without being directly connected to each other. The component that produces the data sends them to a message queue and the consumer checks the queue for messages and receives the data when it detects their arrival. Most message queues have limitations on the amount of data contained in a single message and on the pending number of messages lingering on the queue. A typical message queue also involves the use of message-queueing software, often called a message broker, which manages the queue and stores its state and data. As we present in Sections 4.2 and 5.2 we use this technique in our architecture for the communication of our components.

2.2 ZeroMQ

ZeroMQ is an embeddable library that provides tools for high-performance, asynchronous messaging between components of distributed and concurrent systems. The basic tool it offers is a message queue, which can connect the components and receive and store messages exchanged between them. One particular advantage over other message-queueing software is that there is no need for a separate message broker module [7]. In our distributed architecture we use ZeroMQ for the communication of our components.

2.2.1 ZeroMQ Technology

The ZeroMQ message queues are available through an API that provides ZeroMQ sockets, a generalization of normal TCP sockets, to connect the communicating components through an

endpoint, following a many-to-many communication paradigm. For that purpose, ZeroMQ implements the ZeroMQ Message Transfer Protocol (ZMTP), which defines the way to establish and maintain a connection between ZeroMQ sockets. From the ZMTP specifications derive several types of sockets, that come in pairs defined by the messaging pattern they follow.

ZeroMQ uses background threads to handle I/O to and from the queue asynchronously, which use lock-free data structures to communicate with the foreground threads. It supports many different message transports, such as TCP, multicast (PGM), inter-thread communication (ITC), and inter-process communication (IPC).

2.2.2 ZeroMQ Message Handling

A message in ZeroMQ is handled as A Binary Large OBject (BLOB), regardless of its size. ZeroMQ provides the ability to send multi-part messages, consisting of multiple frames, each holding a separate set of data. The difference from a single-framed message is that, at the end of each frame, there is a "more" bit set to one. The last frame of the message has this bit set to zero. New messages that cannot immediately be received are pushed close to the receiver before they are automatically queued.

Each ZeroMQ queue has a "high water mark" property, which defines its size. When the number of queued messages reaches this number, the queue becomes full, and the way new messages or senders are handled changes, according to the type of the messaging pattern used. In some patterns, all senders are blocked while in others new messages are discarded.

Every message, single-framed or multi-part, is delivered as a whole, as a single message on the wire, in exactly the form it was sent, or not at all. This is possible because ZeroMQ implements strong rules to the exchange of messages. A message will start to be transmitted through the wire only after the last part of it has been sent by the user, and it will only be received by a socket after the last part of it has arrived. In the meanwhile, ZeroMQ stores all parts of a message to be transmitted in memory, until the last one arrives and they are all sent as one.

When the first frame of a multi-part message contains the address of the receiver, we say that the message is wrapped in an "envelope". An envelope is a safe way to include an address to a message, useful for routing messages among possible receivers, without affecting the data of the payload. In this way, we can create general purpose intermediaries, that create, read, add, and remove addresses, regardless of the message structure.

2.2.3 The ZeroMQ Message Transfer Protocol

The ZeroMQ Message Transfer Protocol (ZMTP) is a transport layer protocol which provides specifications for the establishment of a connection between two peers, as well as the exchange of messages between them, when communicating using a connected protocol, such as TCP. The issues it deals with consist of transmission of messages satisfying the TCP limitations, version detecting, security protocols using authentication and encryption and connection metadata exchange.

Two sockets implementing ZMTP are expected to follow a sequence of particular steps in order to establish a connection and exchange messages:

- They start by exchanging data, in order to specify the version and security mechanism of the connection.
- After an agreement on the security mechanism is reached, they initiate its handshake process.
- In the case of successful completion, they exchange metadata about the connection.
- At this point, they can start exchanging application messages.

During these steps, each of the peers is free to abandon the procedure and close the connection at any time.

ZMTP, in general, does not assign specific roles to the sockets, like client and server, but is a peer-to-peer protocol. Some security mechanisms though, do make that distinction, through an "as-server" field included in the greeting, set to 1 for servers and to 0 for clients. This is done in order to assign the task of authentication to servers. If such distinction is not made both peers have the "as-server" field set to 0.

The metadata exchanged during the connection initiation are in the form of a key-value dictionary and consist of two properties:

- The **socket-type**, which determines the type of the sender's socket and may be any of the available socket types.
- The sender's **identity**, used for routing, which may be included when a REQ, DEALER, or ROUTER socket connects to a ROUTER. Otherwise an empty string is sent.

2.2.4 Messaging Patterns of ZeroMQ

The messaging patterns in ZeroMQ define the structure and topology of the system we create. The basic messaging patterns are specified by the ZMTP and are the following:

- *Request-Reply*: This pattern is used in service-oriented architectures, where a group of clients connect to a group of servers and invoke remote procedures on them. There are two different versions of this pattern, the synchronous and the asynchronous, which can be intermingled. The associated socket types are the following: REQ, REP, DEALER, ROUTER.
- *Publish-Subscribe*: In this pattern, there is usually a large number of subscribers connected to a small number of publishers, but the opposite is also possible. The subscribers receive the data the publishers produce, filtered according to their subscription preferences. A common use case for this pattern is event and data distribution. The sockets of this pattern include the publishers PUB and XPUB, and the subscribers SUB and XSUB. The XPUB type can also receive messages from its subscribers and the XPUB can send messages to its publishers.

- *Pipeline*: This pattern typically involves a pipeline consisting of several stages and loops, where nodes push tasks to other nodes, that act as workers, which in turn push their results to nodes deeper into the pipeline. It is therefore intended for task distribution and result collection. This pattern's socket types include the PUSH and PULL types, with the first sending and the second receiving messages.

- *Exclusive Pair*: This kind of pattern has very specific use cases and is usually employed for inter-thread communication within a process. It connects exactly two sockets in an exclusive pair. There is only one type of socket for this pattern, the PAIR type.

Below, we will examine in more detail the request-reply pattern, along with the associated sockets, as it has been proven very useful for the purposes of our distributed system.

2.2.5 The Request-Reply pattern

As we have already discussed, this pattern assigns the roles of the client and the server to the sockets involved. It also distinguishes between synchronous and asynchronous communication. Based on these principles, we derive four different socket types that together form the ZeroMQ request-reply pattern. These are the REQ, REP, DEALER, and ROUTER sockets.

The general idea of this pattern is that a client socket connects to a server socket and sends messages requesting a service. The server processes the request, which usually results in a remote procedure call, and sends a reply back. The messages exchanged in this pattern are always multi-part and have a specific format, with the first frame of each message sent through the wire being a delimiter, an empty frame. This delimiter is used to separate the head of the message, which may consist of several layers of REQ socket identities, used for routing, from the actual payload.

As we present in Section 5.2 we use this ZeroMQ pattern in our architecture for the communication of the broker with the workers.

The REQ socket type

The REQ socket acts as a synchronous client. Its communication pattern consists of sending a request for a service or a set of services and waiting for their replies in a synchronous, lock-step, round-robin fashion.

In more detail, a REQ socket can connect to an arbitrary number of REP and ROUTER sockets, that act as servers. As it is synchronous in its communication, it can only send one message at a time, since it always has to receive a reply for each request, before it can send another. If it is connected to multiple servers, its requests will be distributed to all of them using a round-robin algorithm, and a reply will be expected for each request, from the appropriate server, with the overall message flow consisting of consecutive pairs of request-reply messages.

REQ sockets are the ones responsible for prepending a delimiter, important for the request-reply pattern, in front of all outgoing messages, which is done automatically. Upon sending, the REQ socket type will block if there are no available peers and will not drop messages

it cannot send. On the other hand, it will accept a message only from the last peer it sent a request to and will discard all other messages.

The REP socket type

The REP socket acts as a synchronous server. Its communication pattern involves receiving a request from a client socket, usually passing it then to an underlying application which processes it accordingly, and replying back to the same client with a message. Therefore, its common use case is in systems employing remote-procedure calls. In correspondence with the REQ case, the messaging pattern of receiving a request from a client and then replying back to the same client cannot be broken, for example by receiving a second request before replying, or by sending a reply to a different client.

This type of socket can be connected to any number of REQ and DEALER sockets, which act as its clients, without breaking the single receive - single reply pattern. It receives all messages coming from its clients, without applying any filtering on them or modifying the payload. The format of the messages consists of zero or more identity frames, which are used for identifying the sender, a delimiter, to separate them from the actual message and from one or more data frames.

It receives the messages coming from multiple clients using a fair-queuing strategy. This means that it maintains one queue per connection and serves them in rotation, ensuring that messages from all its clients get processed, regardless of the rate of their message flow. Upon receiving a message, the REP socket removes and stores the envelope, which consists of any number of identity frames, and the delimiter, and passes the rest of the message to the calling application for processing.

Before receiving any other messages, it must send a reply back to its last sender. To do so, a message must be provided by its calling application. The socket then prepends the envelope and the delimiter to the outgoing message and sends it to its appropriate recipient. In case the recipient is no longer available, the socket does not block, but either drops the message silently or returns an error.

The DEALER socket type

The DEALER socket acts as the asynchronous equivalent of a REQ socket. This means that, like the REQ socket, it can be connected to any number of REP and ROUTER sockets, with which it exchanges messages using a round-robin strategy, but it does not need to follow the single request - single reply pattern. On the contrary, it can send and receive messages from any available peer in any order. This is possible by creating and maintaining a double queue for all connected peers, one for incoming and one for outgoing messages. It processes the incoming messages using a fair-queuing strategy. Similar to the REQ socket, it blocks on sending to an unavailable peer, or returns an error.

The ROUTER socket type

The ROUTER socket, in turn, is the asynchronous equivalent of a REP socket. It may be connected to any number of REQ, DEALER, and other ROUTER sockets and exchange messages in any order, using a double queue, similar to the DEALER socket. To identify its

peers, in order to be able to route messages to specific recipients, it uses a unique identity string for each of its double queues. This identity can either be automatically chosen by the ROUTER socket or specified by the peer it identifies.

The incoming messages are received using a fair-queuing strategy. When receiving a message, the ROUTER socket prepends to it a frame containing the identity of the queue from which it was received and passes the resulting message to the calling application. In order to send a message, it removes its first frame and uses its content as identity for its outgoing queues. If a queue with the specific identity exists and is not full, it routes the rest of the message through it, to the appropriate recipient. In case such a queue does not exists, it either discards the message or returns an error, without blocking.

2.2.6 ZeroMQ Example

The steps needed to be followed in order to exchange messages using ZeroMQ in a distributed application can be easily presented and explained using a simple example. For this purpose, we will implement the request-reply pattern using a REQ socket connecting to a ROUTER. We will use function names defined for C language syntax. The example codes are presented in Listings 2.1 2.2

```
int main() {
    ctx = zmq_ctx_new();
    socket = zmq_socket(ctx,ZMQ_REQ);
    zmq_connect(socket,"tcp://localhost:5555");
    zmq_send (socket, "ping", 5, 0);
    zmq_recv (socket, buffer, 10, 0);
    zmq_close (socket);
    zmq_ctx_destroy (ctx);
    return 0;
}
```

Listing 2.1: An exchange of messages with ZeroMQ - Component A

```
int main() {
    ctx = zmq_ctx_new();
    socket = zmq_socket(ctx,ZMQ_ROUTER);
    zmq_bind(socket,"tcp://*:5555");
    zmq_recv (socket, buffer, 10, 0);
    zmq_send (socket, "pong", 5, 0);
    zmq_close (socket);
    zmq_ctx_destroy (ctx);
    return 0;
}
```

Listing 2.2: An exchange of messages with ZeroMQ - Component B

1. As a first step, each of the two components trying to communicate must create a "ZeroMQ context" calling `zmq_ctx_new()`. This function returns a pointer to a context instance, which is the container for all sockets in a process.

2. Next, each component creates its socket instance, by calling `zmq_socket()`, passing as arguments the pointer to our context and the type of the desired socket, and getting a pointer to a socket instance returned. In our example we specify `ZMQ_REQ` and `ZMQ_ROUTER` types for the sockets.

3. Then, we can connect our two sockets calling `zmq_bind()` to bind the first socket to a local endpoint and `zmq_connect()` to connect the other socket to that endpoint. Both take as arguments the appropriate socket pointer and the endpoint, as a string, in the form: `transport://address:port`. Typically, the socket acting as server "binds" and the client "connects" but the reverse is also possible. The supported transports, are tcp, ipc, inproc, pgm, and epgm.

4. At this point we are able to exchange messages using our sockets. In our example, the exchange must be started by the REQ socket, which sends a message to the ROUTER. We can send a message calling `zmq_send()` and passing our REQ socket, our message and any additional flags.

5. On the other end, the ROUTER socket must be able to detect and receive the message. This can be done either with a blocking call to `zmq_recv()`, until the message arrives, or with the help of `zmq_poll()`, used in a loop. This function can be used to poll the desired sockets for activity and notify the caller whenever there is a new message. The caller may then receive the message using `zmq_recv()` and act accordingly.

6. While the ROUTER processes the message, the REQ socket cannot send or receive any other messages, except for a reply back from the ROUTER. This could also be done with a simple blocking `zmq_recv()`, but that would block the whole calling application. Instead, in order to be able to perform other tasks too, we can use `zmq_poll()` again, to periodically check our queue for messages.

7. When a reply from the ROUTER socket finally arrives our REQ socket may send a request again, with the same pattern being repeated throughout the application.

8. Finally, whenever we want to close the connection and terminate our application, we first have to close our socket using `zmq_close()` and then destroy our context calling `zmq_ctx_destroy()`.

2.3 Representational State Transfer

Representational State Transfer (REST) is an architectural style that provides guidelines for the development of Web services. Conforming to REST principles is considered to lead to scalable distributed systems, exhibiting increased performance and greater maintainability [3] [4] [5]. In our architecture we use REST to design our API, as described in Section 5.1

As a service, we consider a mechanism responsible for the management of a collection of system resources and for the presentation of their functionality to users and applications. The service enables access to these resources by exporting a well-defined set of operations using an appropriate interface. A Web service is a service provided over a network and is usually accessible through serialized messages conveyed using HTTP.

Web service APIs designed to conform to REST constraints are called RESTful APIs and they usually have the following properties:

- They use a base URI to provide access to their resources.
- Access to resources is granted through standard HTTP methods (GET, POST, PUT, DELETE).
- They send and receive data using Internet media types, such as JSON.
- They use hypertext links to reference state and related resources.

A REST-compliant Web service applies the following constraints to its components:

- Client-Server Architecture
- Uniform Interface
- Stateless
- Cacheable
- Layered System
- Code on Demand (optional)

Client-Server Architecture

The service is assigns two different types of roles to its components, the client and the server. The clients ask for the service provided and get back replies, while the servers are those who provide the service, receive the clients' requests and reply to them accordingly. In addition, the interface of the service should distinguish clients from servers, which means that each of these groups should not be concerned with the internal issues of the other. In this way, client code becomes more portable and servers more scalable.

Uniform Interface

The uniform interface constraint simplifies and decouples the architecture and enables each part to be developed independently. A uniform interface should be designed following the next four principles:

- *Resource-Based*: Clients use URIs as resource identifiers to refer to individual resources in their requests. In addition, servers do not return the resources requested as they are internally processed and stored, but a representation of them using data serialization formats, such as XML or JSON.
- *Manipulation of Resources through Representations*: The representation of a resource sent to a client includes all the necessary information the client needs, in order to modify or delete the resource on the server, as long as it has such permission.

- *Self-descriptive Messages*: Each message should contain all the information necessary to describe how to process it. Responses, especially, should also indicate their cacheability.
- *Hypermedia as the Engine of Application State (HATEOAS)*: This constraint means that client or server state is exchanged through hypermedia. Clients in particular use body contents, query-string parameters, request headers and the URI identifiers, while servers deliver state or URIs for resource retrieval via body content, response codes, and response headers.

Stateless

Statelessness demands that all session state is kept entirely on the client. That means that each request from client to server must include in the body all the necessary state information to process it, without using any state stored on the server. Respectively, the header, status and body included in each server response must contain all session state data. This constraint enables greater scalability, since the server is concerned only with the resource state, which is the same for all clients. On the other hand, each client must store all data relevant to its particular session.

Cacheable

Typically, in Web services, clients are able to cache responses, in order to re-use the information provided in the future, avoiding additional interaction with the server, thus increasing scalability and performance. It is therefore important that responses specify, either implicitly or explicitly, whether they are cacheable or not, to prevent clients from caching invalid information.

Layered System

According to this constraint, a client has no means to verify if the server it is connected to is the actual end server or an intermediary, located in between. Intermediary servers sometimes are used to apply security policies and increase scalability, by providing load-balancing and shared caches.

Code on Demand

Code on demand allows servers to pass to clients logic they can execute, thus affecting their functionality. Logic can be transferred by compiled components such Java applets and client-side scripts such as JavaScript.

These constraints provide distributed Web services with properties such as:

- Performance by accelerating component interactions and thus user-perceived performance
- Scalability by enabling them to support large number of components and interactions between them

- Simplicity of interfaces
- Modifiability of components
- Visibility of communication between components
- Portability of components
- Reliability by making them resilient at system level to individual component failures

2.4 Django REST framework

Django REST framework is a toolkit, written in Python, that provides an easy way to create RESTful APIs for database-driven applications. A Web framework, in general, provides the necessary programming infrastructure for creating applications without having to develop everything from scratch, focusing on code reusability and component pluggability. Django follows the Model - View - Controller (MVC) architectural pattern. In this thesis, we use Django to implement our RESTful API.

Model - View - Controller

MVC is a software architectural pattern, according to which a software application is divided into three loosely coupled parts, in order to distinguish between the internal representation of data and the form in which they are presented to users. In addition, each part can be developed and modified independently, without affecting the rest of the application. The distinct parts are described below:

- *Model*: This component is responsible for defining, managing and accessing data, independent of the user interface.
- *View*: This is the component that decides what data should be displayed and the way they should be represented.
- *Controller*: This component receives user input, accesses any models, if needed, and selects the appropriate views to present.

In Django, these components are handled in the following way:

- The Model is handled by the database layer. This layer includes Django models, which consist of Python code describing our data in the database. A model defines everything about the data, including its behaviour, the way to access and validate it, and its relationship with other data. Models are used by Django to execute SQL queries in the background and return database records using Python data structures.
- The View is handled by views and serializers. A Django view is a Python function that takes an HttpRequest as parameter and returns an instance of HttpResponse, using the appropriate models and their serializers to perform SQL queries to the database. A serializer is code which enables the conversion of complex data such as model instances

or query sets to native Python datatypes which can in turn be rendered into content types such as JSON. They also provide deserialization, the conversion of parsed data into complex types, after their validation.

- The Controller is handled by the framework itself, through a URLconf. A URLconf maps URLs to the view functions that should be called when they are accessed.

Chapter 3

Existing Evaluation System

The initial form of the evaluation system was developed using the C programming language and ran on a single computer receiving submitted solutions and evaluating them sequentially.

3.1 Architecture

The architecture of the system consists of two layers: A front-end Web server and the back-end evaluation system. In also includes a database where crucial data are stored.

The Web server provides an interface to users through which they can access the grading resources. It receives their submissions and passes them to the evaluation system for grading. In more detail, once a new submission arrives, the server stores it in the database, where it acquires a unique id. The server uses this id to store the necessary data concerning this submission to a queue in the local files of the evaluation system.

The evaluation system reads new submissions from its local queue and proceeds to compile, execute and evaluate them. The general architecture is presented below (Fig. 3.1).

To organize its data, the system uses several entities. These are:

- Tasks
- Submissions
- Tests
- Graders

3.1.1 Tasks

A task provides the specifications of the problem for which solutions are submitted. These include:

- the **input** and **output** files of the solutions
- the **time limit** and the **memory limit**

- the **number** and **id**s of the tests used to evaluate the solutions
- a special flag field indicating default or custom grading
- a flag field specifying whether the solutions are interactive programs or not

There are three types of tasks:

- batch
- reactive
- output only

Batch tasks make up the vast majority of the tasks that appear in programming contests, which is why we focus particularly on them. The existing evaluation system provides support only for the first two types. The system we designed and present in this thesis currently only supports the first type. However, it is easy to integrate grading for the other two types of tasks in the future.

3.1.2 Submissions

A submission is the source code, along with the necessary metadata, submitted as solution to a particular task.

- Received submissions are placed in a queue, from which they are retrieved for grading.
- The metadata include the name of the relevant task, the time and memory limits, the input and output filenames and the ids of the test cases to be used for the evaluation process.
- Submissions are identified by an incremental counter, which is tracked by the client.

3.1.3 Tests

These are the test cases used to evaluate the submitted solutions. Each test consists of an input file which is read by the submitted programs. These programs then produce an output file which is compared with an expected output, given for every test case, to check whether the two files match byte by byte. For specific tasks, where a different form of evaluation is needed, a custom grader is used which is provided by the client (see next paragraph). There is a set of available test cases for a given task and the client must specify a subset of these to be used for the evaluation of a particular submission.

3.1.4 Graders

A grader is the program used for grading the submitted solutions based on the output they have produced. There is a default grader, embedded in the system, but there is also the capability to use a custom grader, defined by the user.

- The default grader simply compares the output of the program for each test with its expected output to determine whether it is correct or false. To use the default grader the task has to specify it in the appropriate flag of the submission.
- A custom grader may work differently and has to be provided by the user, in code. For example, the former default grader is not helpful in cases where multiple solutions to a problem may exist, such as the shortest path in a graph, and a different way to evaluate the solutions has to be employed. Custom graders take the output of the submission and the expected output for each test case as input and grade the submission with a decimal number in the range [0,1].

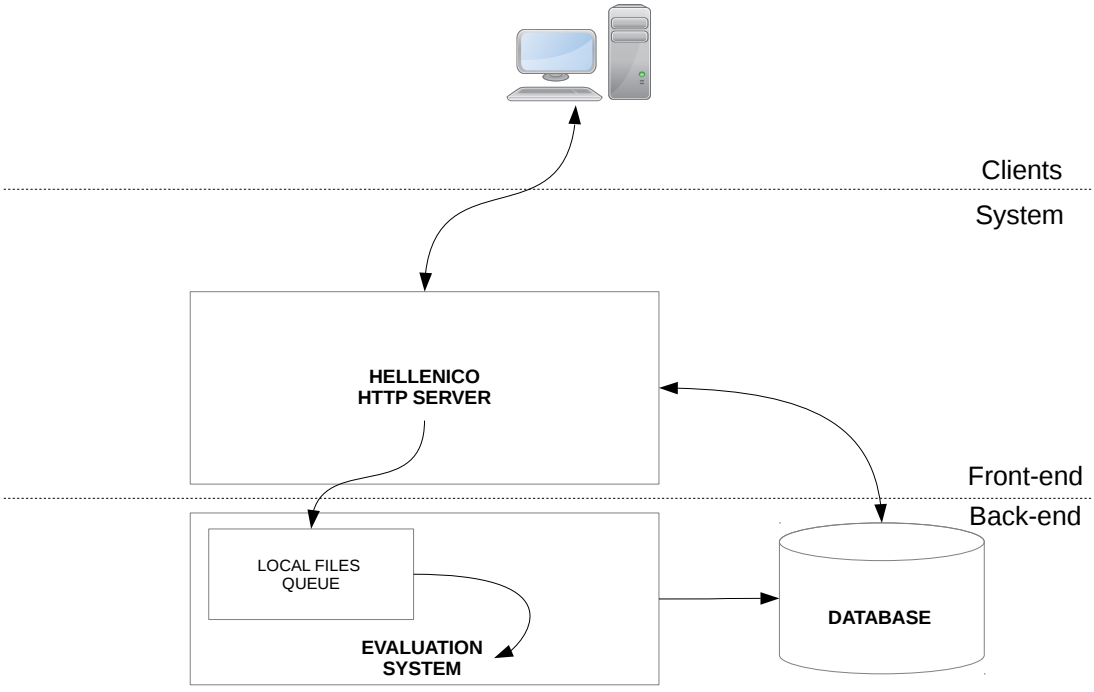

Figure 3.1: The overall architecture of the original evaluation system

3.2 Behaviour

The overall behaviour of the system consists of three phases:

Receive and Prepare

In this phase the Web server receives a new submission. The submission is then stored in the database, which assigns a unique id to it. The server receives this id and uses it as submission identifier when it adds it to the grader's queue.

The grader keeps the expected id of the next submission to be evaluated in a file. The id in the file is incremented once the corresponding submission is graded. In order to process the next submission in the queue, the system reads the expected id of the next submission from the file and periodically checks if the files associated with this submission exist in the queue. When it finds them, it opens them, reads all the necessary information and initiates the process of evaluation.

Compile

During this phase, the system reads the source code of the solution and compiles it according to the programming language used, linking any necessary code, if the related task is reactive. In case of unsuccessful compilation, the compile errors are stored in a file and the evaluation process terminates.

Run and Grade

After a successful compilation the system runs the resulting executable once for each test case, providing it as input, and stores the output it produces in a file for grading. Once the output for a test is produced, the system evaluates it, using the appropriate grader. After evaluation, the grader stores the results in a database, where the Web server has also access. It then sends an empty request to a callback URL, unique for each submission, to notify the server that the results are ready, in order read them from the database and make them available to users.

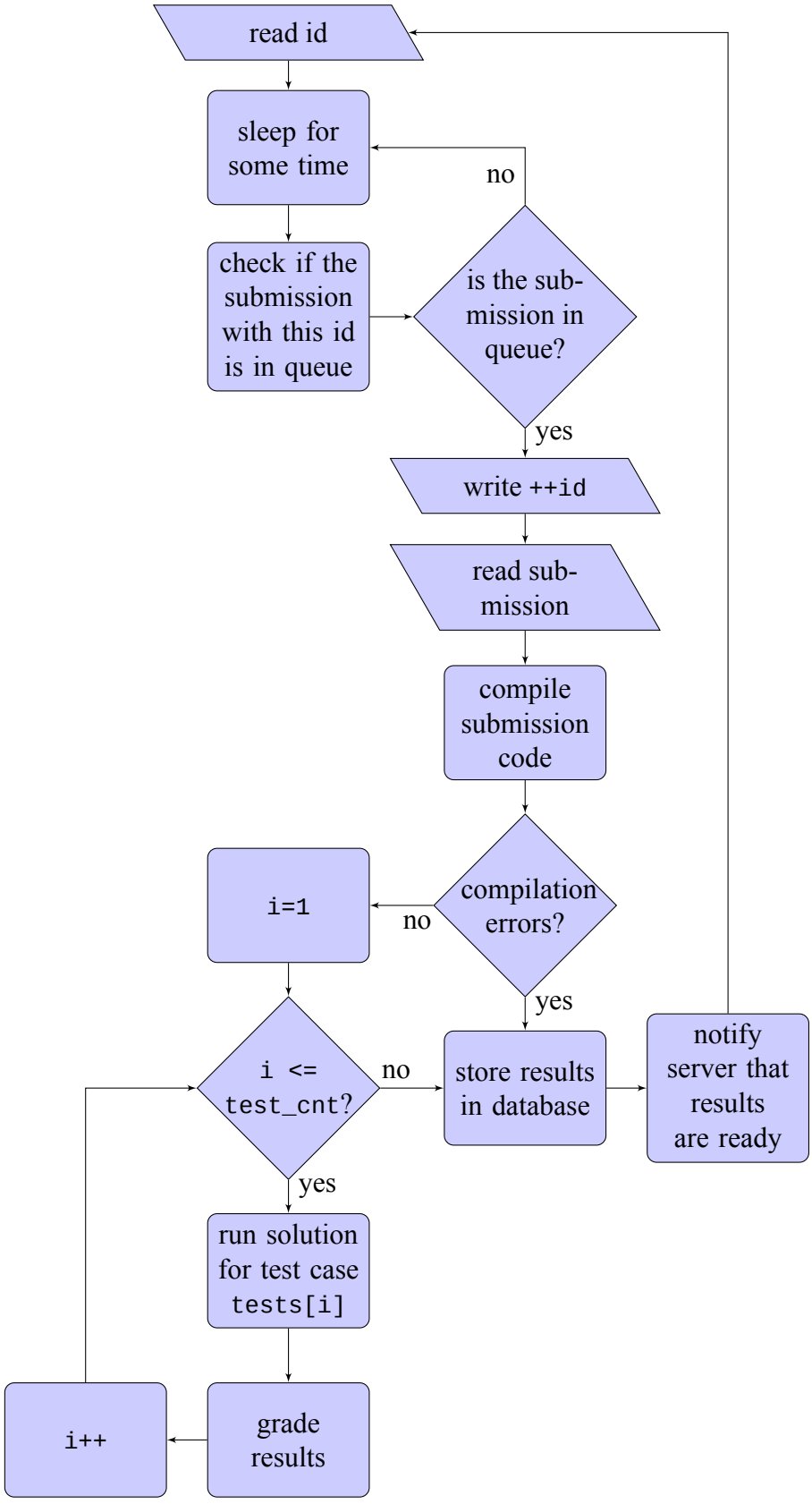

Figure 3.2: The overall behaviour of the original evaluation system.

Chapter 4

Design

In this chapter we describe the approach followed during the design of our evaluation service. The first goal is to use the existing system in order to create a database-driven distributed application with the following properties:

- *Scalability*: To provide this property the grading system needs to be able to run in parallel on as many nodes as there are available, without affecting performance. The aim is to create an application which will distribute work to these systems, exploiting the parallelism created.
- *High Availability*: This property is ensured by eliminating single points of failure when possible, providing the system with the ability to withstand partial failure of a subset of its components. In such cases, other parts of the system will offer the functionality of the lost part, guaranteeing that the downtime of the service is minimized.
- *Fault Tolerance*: To fulfill this target, our architecture makes certain that failures will not lead to loss of submissions or other crucial data and the system will be able to fully recover to its previous state after restarting.
- *Load balancing*: In order to increase its performance, a scalable application needs to be able to distribute the workload equally among its resources. To achieve this, our system needs a mechanism to dispatch submissions to all the concurrently running grading systems, using a fair strategy, thus avoiding overloaded queues on the one hand and idle graders on the other.

The second goal is to create a REST-compliant service to provide access to our grading resources publicly.

4.1 Design Overview

To achieve our goals we created a topology for the service, where the nodes are divided into two major groups:

- *The HTTP servers*, which comprise the front-end of the service. They implement and expose a RESTful API and accept any requests through HTTP. If the requests are submissions for evaluation they are issued to a message queue, so as to be processed by the evaluation systems.

- *The back-end workers*. These consist of the evaluation systems running in parallel. They receive submitted tasks from the queue, evaluate them, and handle the results based on specifications provided by the client.

Between them there is a message broker that connects them, ensuring that messages from the HTTP servers are passed to the workers for processing. Connected to the broker, and communicating only with it, is a safety system used for fault-tolerance. In addition, the system includes a database, for storing necessary data, as it is explained below. Access to the database is granted only to the front-end, the broker and the safety service.

With the exception of tests, all the rest information is stored in the database, via standard SQL queries. Tests were decided to be stored in a distributed file system, so as to be easily accessible by large numbers of workers, which are designed not to have access to the database.

The new architecture is presented below (Fig 4.1).

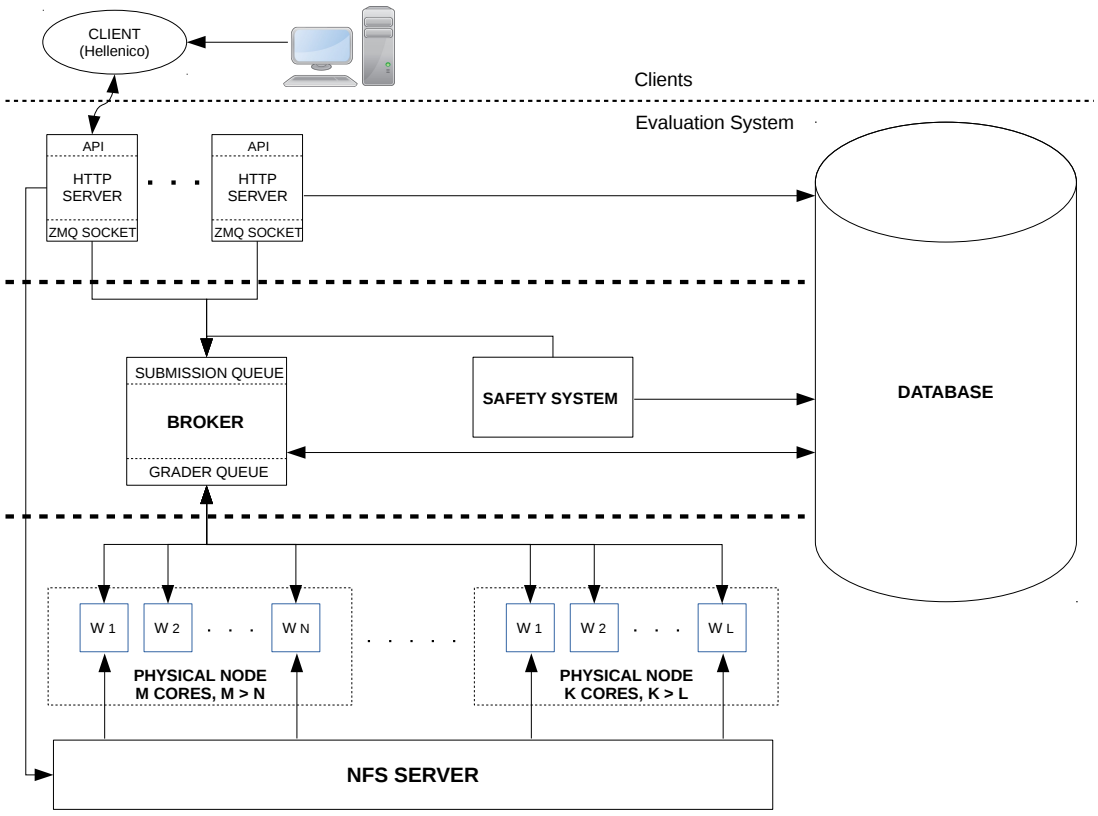

Figure 4.1: The proposed distributed architecture for the evaluation system

4.2 Design Details

In this section we provide details about the design of the system components mentioned above as well as their behaviour and the interactions among them.

4.2.1 HTTP servers

In more detail, the HTTP servers on the front-end expose a RESTful API, which provides a series of operations to clients, such as the ability to perform CRUD operations on tasks and their test cases. This is done in a transparent way, via SQL requests that the servers perform on the database, where most data is stored, in the background. Also, the API offers the ability to submit solutions to particular assignments and a way to check the status of each submission. Additional functionality, such as listing all the saved assignments, tests, graders and preparing the graders for intensive work by fetching all the needed data to memory, are also supported. An elaborate description of the API is provided below. Finally, they receive the results for each particular submitted solution by the workers and update the database.

4.2.2 Broker

Of all the requests to the service, only submitted solutions are passed to the back-end. All the rest functions are provided by the HTTP servers. Submitted solutions are received by the servers and sent to a queue, waiting to be processed. The broker's job is to assign these new submissions to available workers in the back-end. This is done using a client-server pattern. Whenever a worker is available, it sends a request to the broker asking for work and the broker replies with a submission from the queue. The broker also checks and updates the database to ensure that all submissions are processed. We will explain this procedure in detail in Section 4.5

4.2.3 Workers

The back-end is comprised from a set of workers, each of whom consists of a copy of the evaluation program, running on a computer node. Each node runs multiple such copies and is part of a cluster which includes multiple nodes. The workers ask the broker for work, whenever they become idle, thus providing an even distribution of the load to the system. Each worker is an independent, pluggable component, and thus workers can be added or removed at will, depending on the resources available and the existing workload. When a worker finishes evaluating a submission, it sends the results to the HTTP servers using a standard, predefined URL. Clients, however, have the capability to specify another URL of their own, where they want their results to be sent after grading.

4.2.4 Safety System

The safety system is a program running on a node, whose job is to keep track of the database for any submissions that have been being processed for a particularly long time, indicating that something went wrong along the way (possible a worker failure). Whenever it finds such a submission, it should send it back to the queue for processing, marking the time it was last modified, so as to avoid sending it over and over again.

4.2.5 Database

Our application needs to handle a lot of data coming from clients, which should be safely stored and easily accessed. Therefore, we decided to employ a database-driven architecture and use a database for that purpose. Below we present the design of our database (Tables 4.1 - 4.4), as it derives from the needs of our application.

Field	Type	Notes
`id`	`integer`	primary key
`task_name`	`varchar`	unique for each task
`locked`	`boolean`	if locked, task cannot be modified or deleted
`input_file`	`varchar`	the file from which input is read
`output_file`	`varchar`	the file from which output is read
`time_limit`	`float`	the maximum time a solution is allowed to run
`memory_limit`	`integer`	the maximum memory a solution is allowed to use
`grader`	`varchar`	the grader to be used for this submission
`link`	`varchar`	this field is intended to be used for linking external code

Table 4.1: The task table

Field	Type	Notes
`id`	`integer`	primary key, returned to user
`task_name`	`varchar`	the name of the relevant task
`language`	`varchar`	the programming language of the source code
`code`	`longtext`	the source code of the solution
`tests`	`varchar`	the ids of the test cases to be used for this submissions
`callback_url`	`varchar`	optional, user-provided, used to post the results
`django_url`	`varchar`	predefined, server URL, used to post the results
`timestamp`	`integer`	the time a submission last changed its status
`status`	`enum`	the status of a submission, indicating its stage in the system
`compilation_status`	`varchar`	'COMPILEOK' if successful, 'NOCOMPILE' otherwise
`compilation_errors`	`varchar`	the errors when status is 'NOCOMPILE'

Table 4.2: The submission table

Field	Type	Notes
`id`	`integer`	primary key
`grader_name`	`varchar`	a string identifying a grader
`language`	`varchar`	the programming language of the grader's source code
`source`	`longtext`	the source code of the grader

Table 4.3: The grader table

Field	Type	Notes
`id`	`integer`	primary key
`test`	`integer`	the relevant test case id
`run_time`	`double`	the time needed to complete the execution of the program
`score`	`double`	the score achieved for this test case, in the range [0,1]
`output`	`varchar`	the output produced for this test case
`expected_output`	`varchar`	the correct output for this test case
`sub_id`	`integer`	the id of the relevant submission
`result`	`enum`	a string describing the result for this test case

Table 4.4: The result table

4.2.6 Cache

Since we store the test cases in a distributed file system, we want to be able to bring them to a worker's local storage, in order to decrease their access time and increase throughput. This is useful in situations, such as a competition, where a large number of solutions is submitted for a set of specific tasks and, consequently, is tested using the same set of test cases. Therefore, we provide two API calls, "fetch" and "release", to manually fetch the necessary test cases to the workers' local storage and delete them afterwards. An even better solution would be to do this action automatically, but designing a cache is beyond the purpose of this thesis. However, it can be useful future work.

4.3 Scalability

In this section we explain how the proposed architecture provides the application with scalability. More specifically, we describe the way our design enables the system to exploit the capabilities provided by a cluster environment to deploy multiple worker instances in parallel, achieving an increase in its performance proportionate to the resources available.

Our focus on scalability is to provide the system with the capability to process multiple submitted solutions simultaneously. The way to achieve this is by deploying many workers on cluster nodes pulling work from the queue. Our design offers an easy method to accomplish this, by providing pluggable and independent workers that can be added effectively and without a limit to the system, as long as there are resources available. In every worker node, the system deploys workers, the number of which is proportionate to the available cores. After being deployed, workers are connected to the broker and can normally pull work from the queue by sending a request.

Since the workload consists only of submissions, which are independent from each other, and each submission is evaluated by a single grading system, there is no need for synchronization or communication between the workers in our system. Therefore, an increase in the number of workers results in a proportionate increase in the maximum speedup, without any limitations imposed by a bottleneck or communication overhead. Consequently, we can keep adding workers and getting a proportionate increase in speedup indefinitely.

4.4 High Availability

Our system ensures high availability by providing ways to keep the service available, even when failures happen to the front-end or the back-end of the system.

On the front-end, multiple HTTP servers expose the RESTful API and listen to client requests. Consequently, in the event of a failure on an HTTP server node, other servers will continue offering the service, receiving client requests that would otherwise be processed by that server.

On the back-end, multiple instances of the grading system run in parallel, evaluating submitted solutions. When a worker fails, we make sure that its work will be reproduced and resumed by a different worker, using a mechanism we describe in the next paragraph.

Using this approach, we minimize downtime and we provide the clients with the experience of a highly available service.

4.5 Fault Tolerance

We can perceive how fault tolerance is provided to the system by our design, by listing the various types of individual component failures and describing the ways the system will respond to ensure availability. The most common type of failure in cluster environments is node failure. In our application we have several types of nodes, each one of whom may fail independently. These are:

- HTTP servers
- A Broker
- A Safety system
- Workers

4.5.1 HTTP server node failure

When an HTTP server node fails, other HTTP servers may continue the normal provision of the service, or, in the absence of alternatives, it will become temporarily unavailable, denying any new requests to clients, until a server comes online. The key in this scenario is to ensure that no crucial data that will affect the function of the system are lost, as a result of the server failure.

All data sent to an HTTP server are stored, either in a file, like the tests, or in the database, as is the case with the rest of the data. Therefore, loosing any of these data can be easily checked with a simple SQL query to the database and results in a simple repetition of the request that produced them. A problem arises though, when we handle submissions, for which we must ensure that they reach the back-end of the service and get processed, otherwise they may be lost, even after being stored in the database, without the user ever knowing it.

To accomplish this we designed the system to block upon receiving a new submission, until it stores it to the database and sends it to the broker queue. When creating the record in the database, it uses a **PRE_QUEUE** status for the new record, meaning that the submission has not yet been sent to the queue. A submission with **PRE_QUEUE** status is not considered fully accepted yet, as the client has not received a confirmation and an identification, which will enable them to look for the results later.

Therefore, after storing and queueing the submission, it sends a reply back to the client, containing the unique id assigned to the submission record in the database, informing them that their solution has been successfully queued for evaluation. In this way, if the client does not get a reply back, they will know that their submission failed at some point and they have to send it again. On the other hand, receiving a reply means that the submission has at least reached the queue and it is up to the back-end of the system to ensure its successful evaluation.

Afterwards, it updates the submission status in the database as **IN_QUEUE**, which signifies the successful completion of the queueing process. Submissions that have reached **IN_QUEUE** status are guaranteed by the system to complete their evaluation.

4.5.2 Broker node failure

A broker failure would mean that, while the broker is offline, neither submissions waiting on the queue nor any new ones would be able to get processed, since the broker is the manager of the queue. However, our design makes it easy to restore the broker's previous state, after a failure, and ensures that no data are lost because of it. Furthermore, the broker, like most of our system's components, is an independent and pluggable component, being connected with the rest of the system only via message queues. Therefore, restarting it is practically costless, and its failure hardly affects the overall performance. This becomes clear by examining what happens when the broker node fails.

As we have seen, when a new solution is submitted by a client, it has to be stored in the database and sent to the broker's queue, before it is considered successfully submitted. Therefore, while the broker is offline, any new submission requests block, without failing on the client's side, until the broker restarts and queues them. Of course, a client may quit and try again later, by re-submitting their solution, but the broker will receive and queue both submissions. As a result, a broker failure only adds some user perceived latency to the system, until the broker restarts.

After restarting, the broker needs to restore the state of its queue before the failure. The initial queue was lost during the failure, but the submissions are stored in the database and can be retrieved. For that purpose, the broker is provided with a mechanism to recognize which submitted solutions were on its queue when the failure happened, and get them from the database, in the right order. Using this method, we ensure that a submission that has been successfully queued, won't be lost because of a broker failure.

This mechanism is based on the **status** attribute of each submission. Only submissions with **IN_QUEUE** status are waiting in the queue managed by the broker. Therefore, the broker will ask from the database only the submissions with this status and will put them back to its queue to be processed. Upon sending one of them to an available worker, it will update the submissions' status in the database as **PROCESSING** and its timestamp to current time.

The overall behaviour of the broker is presented in Fig. 5.1

4.5.3 Safety system failure

A failure on this node does not pose any danger to the system or affect its performance, as it is not crucial to its function or holds any important data. The only way a failure in this system can affect the overall behaviour is if it happens while it tries to send requests from the database back to the queue. In that case, it will restart and try to send them again in the future, after re-checking the database for "old" submissions. Therefore, it may add some form of latency, which can be minimized by checking the database at short intervals.

This system will check the timestamp of each submission with status **PROCESSING**, as it looks for failures happening on the worker nodes.

4.5.4 Worker node failure

Worker nodes do the most important and heavy work in our system and, therefore, they are considered the most unreliable and prone to failures. For this reason, we designed them to be expendable and their individual work easily reproduced by other peers.

In more detail, each worker connects to the server and asks for a solution to evaluate. After it receives a submission, it evaluates and grades it and sends the results to their designated receiver. At any point during this process it may fail, with the results being unpredictable. In our system, all worker failures are handled in the same way. However, there are two types of failures that seem to have different effects:

1. A failure while a worker is idle, waiting for work.
2. A failure while a worker is evaluating a submission.

We will explain how these failures result in the same state and therefore are treated as one by our application.

In our design, we do not store any state for a particular worker nor we restart workers to continue their work. When they fail we just create new instances of workers that replace them. Consequently, the new instances are completely oblivious to their predecessors' state. Simply put, a failing worker's work is lost with it. However, this does not mean that we violate our principle that all submissions are eventually processed and evaluated. After some time, the safety system will notice that a particular submission takes too long to be graded, meaning that a worker has probably failed, and it will send it back to the queue to be processed by a different worker.

Our system is designed to exhibit the same response when the failing worker is idle. This results from the fact that the broker keeps a list of the available workers, without checking their liveness, meaning that an available worker may fail, without the broker knowing it. This, along with the fact that we do not restart workers, but create new instances instead, leads to the broker eventually sending a submission to a non-existent worker, meaning that the particular copy of the submission is lost. However, it will be later re-queued and finally

processed when the safety system takes action, as explained in the previous case. Therefore, unreliable workers do not account for an unreliable system, but they add delay to the process of evaluation, which can be controlled using a well designed safety system.

4.5.5 Example

We can easily understand how the system ensures submission evaluation, by overviewing the stages through which a new submission passes (Fig. 4.2):

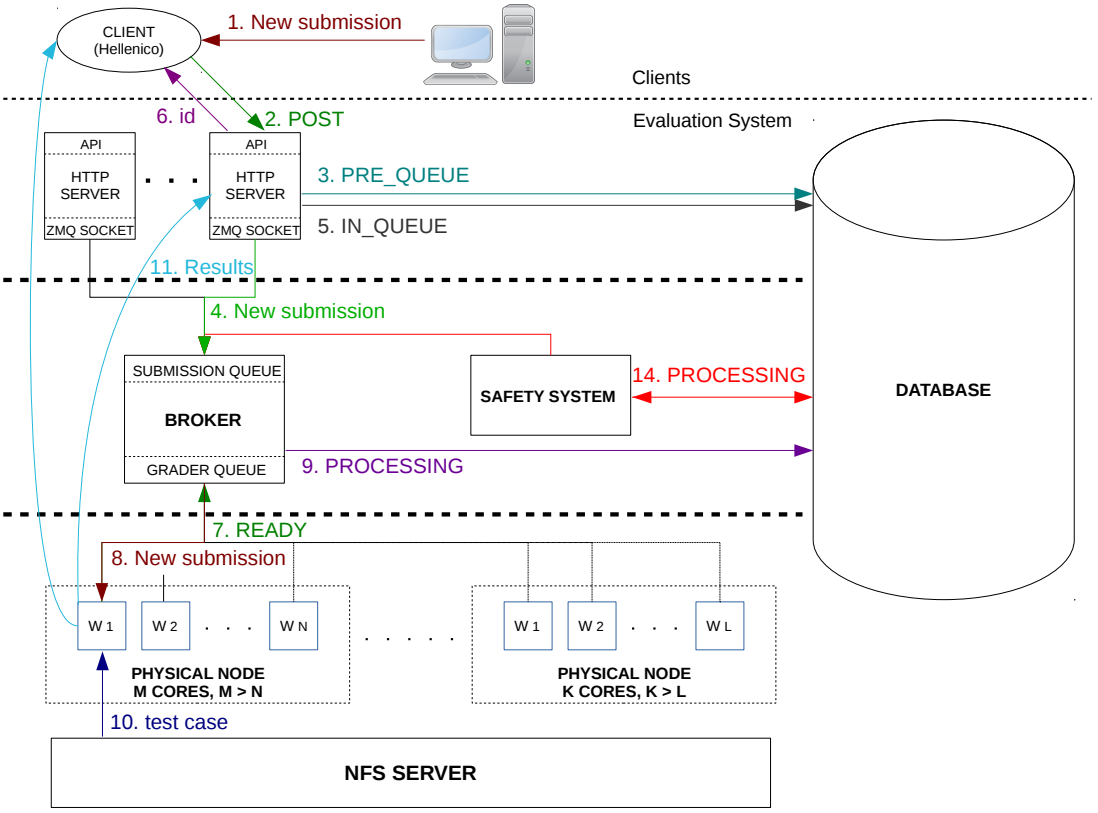

Figure 4.2: The exchange of messages in the system in order to evaluate a submission

1. A new submission arrives to a client of our service.
2. The submission is posted to a front-end server.
3. The submission is inserted into the database by an HTTP server with a **PRE_QUEUE** status, and acquires a unique submission id, which is returned to the server.
4. Then, it is sent to the broker, where it is queued for processing.
5. The HTTP server updates the status of the submission to **IN_QUEUE**.
6. Afterwards, a response message is returned to the client, along with the unique id, necessary to track the submission progress. If there is a failure when receiving, storing or

queuing the submission, an error message will be returned to the client, thus ensuring that only when a request has reached the back-end will it be considered successfully submitted.

7. A worker becomes available and sends a request for work.
8. The broker replies to the worker's request with the new submission.
9. The broker changes the status of a submission to **PROCESSING**, along with its timestamp, right after it has been sent to the worker. In case the broker fails, no new solutions can be submitted and also it is made sure that none already submitted solutions will be lost, because every time the broker is restarted it automatically loads any **IN_QUEUE** submissions from the database , restoring its previous queue.
10. The worker compiles the submission and loads the necessary test cases from the node where they are stored, which is an NFS server.
11. The worker grades the submission and posts the results to the server and to an optional user defined URL.
12. The server stores the results and updates the submission status to **COMPLETED** .
13. The server replies with the results to subsequent requests.
14. The final problem is to handle worker failures and this is done with the help of the submission safety service. Its purpose is to check the database periodically and find any submissions, with status **PROCESSING**, considered "old" based on their timestamp. An old timestamp, along with a **PROCESSING** status, means that a worker has probably failed, leaving its evaluation unfinished. It will then send those submissions back to the broker to be processed again, thus ensuring that no submitted solutions will be left waiting indefinitely.

4.6 Load Balancing

Our design provides the system with load balancing in a very practical and perceivable way, using the request-reply pattern for the workers. All the available work is stored in a centralized, FIFO queue, from which every worker is assigned submissions, whenever they become idle. Using this approach, we ensure that all workers are assigned only as much workload as they can process, avoiding situations where an unequal distribution of work results in overloaded queues in some workers and empty queues in others.

4.7 Summary

In conclusion, our system consists of six components:

1. *The HTTP server nodes*, which receive and process all requests from clients, and reply back to them accordingly. Apart from operations concerning tasks, tests and graders,

which are handled only by the front-end of the service, their main job is to receive new submissions and reply back to the clients after successfully storing them in the database and sending them to the queue. They also receive the results from the workers and update the database accordingly.

2. *The broker node*, who is responsible for managing the submission queue, by sending pending submissions, queued by the HTTP servers, to available workers. It also communicates with the database, to check about submissions that were lost from its queue without being processed and to update submission status. Furthermore, it accepts messages to its queue coming from the safety system, concerning submissions that were lost while being evaluated by workers.

3. *The worker nodes*, that are evaluation systems running on cluster nodes. They ask and receive submitted solutions from the broker, evaluate them, and send the results back to the HTTP servers to be stored in the database, using a standard URL. If specified, they also send the results to a user defined callback URL.

4. *The safety system node*, which checks the database for submissions that failed while being processed and sends them back to the broker queue.

5. *The database*, in which all data, with the exception of tests used for evaluating submitted solutions, are stored.

6. *The NFS server*, which keeps the test cases and shares them with the worker nodes.

Chapter 5

Implementation

In this chapter we describe the means and methods we employed to implement our design and build our distributed service. We start by presenting the implementation of the front-end and continue with the back-end.

5.1 Front-end Implementation

To implement the RESTful API described in Section 4.2 we used the Django REST framework (Section 2.4). We will describe in detail the models we defined for our database representation and the views we created to present these views to clients.

5.1.1 Models

Based on the design of our database we created the following models:

Task

The task model is a class that represents a task, as initially defined in Section 3.1, and later presented for the database design in Section 4.2. A task class model has the following fields:

- *task_name*, a charfield, used by submissions and URIs to refer to a task.
- *locked*, a boolean field. When it is true it is not possible to modify the task in the database
- *input_file*, a charfield with default value “stdin”. Defines the input of the solutions.
- *output_file*, a charfield with default value “stdout”. Defines the output of the solutions.
- *time_limit*, a floatfield. It is the time limit for solutions.
- *memory_limit*, an integerfield. It is the memory limit for solutions.
- *grader*, a charfield, with blank default value, which means using the default grader

- *link*, a charfield, with blank default value, which means that the tasks are not reactive. In case of a reactive task, this field specifies the code to be linked. We should note that, although this field exists, the related feature is not implemented in the existing version of the system, but will be added in a later version.

The unique id for each task and the rest of the models, is not included explicitly in the model class, but added automatically by the database.

Submission

The submission class model, represents a submission as presented for the database design and discussed throughout the previous Chapter (4). The fields this class includes are:

- *task_name*, a charfield that specifies the name of the task, to which this submission is referred.
- *code*, a text field containing the source code of the submission
- *language*, a charfield containing the programming language of the code
- *callback_URL*, a URL field containing an optional user provided callback URL to use for sending the results.
- *status*, a charfield containing the status of each submission. Its range of values is an enumeration consisting of **PRE_QUEUE**, **IN_QUEUE**, **PROCESSING**, **COMPLETED**.
- *compilation_errors*, a charfield, containing any compilation errors, in case the source code provided has errors and cannot compile.
- *timestamp*, an integer field, containing the time that the submission was last modified in the database.

Grader

The grader class represents a custom grader that can be added by the user to grade the submissions to particular tasks and includes the following fields:

- *grader_name*, a char field specifying the name of the grader. It is used by submissions to refer to that grader.
- *source*, a text field containing the source code of the grader.
- *language*, a char field specifying the programming language used in the source code.

Result

The result class model represents a result referred to a particular submission and concerning a specific test. The fields it contains are:

- *sub*, a foreign key. Contains the id of the submission to which the result is referred.
- *test*, an integer field specifying the particular test it represents
- *run_time*, a float field containing the runtime of the submitted solution
- *score*, a float field containing the score gained for the specific test
- *output*, a char field containing the output produced for that test
- *expected_output*, a char field containing the expected output for that test

5.1.2 Views

The views we created, based on our API, to present and modify our models, are the following:

`task_list`

This view implements the *Get Task List* and the *Add Task* operations of the API. It is called using a `GET` or a `POST` method to the `/tasks` URI.

When called with a `GET` it implements the *Get Task List* operation and returns a list containing the tasks stored in the database or a `404` (not found) status code.

When called with a `POST`, respectively, it implements the *Add Task* operation, which expects to find the JSON containing the task specifications in the body of the request. If the information in the body is in the correct form, it stores the new task and returns a response containing a `201` (created) status code along with a JSON with the stored data of the task. Otherwise it returns a `400` (bad request) status code discarding the request data.

`task_detail`

This view implements the *Get Task Info*, the *Modify Task* and the *Delete Task* operations of the API. It is called using a `GET`, `PUT`, or a `DELETE` method respectively, with the appropriate `task_name`, to the `/tasks/<task_name>` URI. If the task with the requested task name is not found it returns a `404` (not found) status code.

When called with a `GET` - *Get Task Info* - it returns a `200` (ok) status code and a JSON containing the data stored in the database for the specific task.

When called with a `PUT` method - *Modify Task* - including the JSON containing the task information to be modified, it updates the new task data and returns a response containing a `200` (ok) status code along with a JSON with the updated data of the task. If the request is malformed it returns a `400` (bad request) code without modifying the task data in the database.

Finally, when called with a `DELETE` - *Delete Task* method, it deletes the specified task from the database and returns a response containing a `204` (no content) status code.

`grade`

This view implements the *Grade* operation. It is called using a `POST` method to the `/submissions` URI. In the body of the request a JSON should be provided containing the necessary information for the new submission, as described in the API. It uses a **DEALER** socket, created by the server, to connect to an endpoint bound by the broker and to send to it received submissions. We use a **DEALER** socket so as to be able to send consecutive requests, without waiting for replies.

The function stores the new submission to the database, with a **PRE_QUEUE** status and then blocks on sending to the broker until the submission is successfully queued. It then updates the submission status to **IN_QUEUE**, as described in chapter 4, and returns back to the client a 201 (created) status code, along with a JSON with the unique submission id, assigned to its record in the database. If the `task_name` provided in the JSON does not correspond to a database record, the view returns a `404` (not found) status code and the submission is discarded. Also, if the request is malformed it returns a `400` (bad request) code.

`results`

This view implements the *Grade_info* and *Post_results* operation. It is called using a `GET` or `POST` method respectively to the `/submissions/<sub_id>` URI, where sub_id is the unique submission id obtained from the body of the `grade` response.

When called with a `GET`, if the submission id exists, this view returns the data stored in the database concerning the evaluation of this particular submission, as presented in the API. If the id provided does not correspond to an existing database record, a `404` (not found) status code is returned.

If a `test_id` argument is provided with the `GET` method after the resource URI, in the form `/submissions/<sub_id>/?test_id=<test_id>`, the view returns a `200` (ok) status code, as well as explicit results concerning the particular test. If there are no results for the test specified a `404` (not found) status code is returned.

`add_grader`

This view implements the *Add Grader* operation of the API. It is called using a `POST` method to the `/graders` URI. The request must include a JSON containing the grader specifications, as defined by the API. If the information in the body is in the correct form, it stores the new task and returns a response containing a `201` (created) status code along with a JSON with the stored data of the task. Otherwise it returns a `400` (bad request) status code discarding the request data.

delete_grader

This view implements the *Delete Grader* operation of the API. It is called using a `DELETE` method to the `/graders/<grader_name>` URI, where grader_name specifies the desired grader. If it exists, the view deletes the specified grader from the database and returns a response containing a `204` (no content) status code. Otherwise it returns a `404` (not found status code).

add_test

This view implements the *Get Test List* and the *Add Test* operations of the API. It is called using a `GET` or a `POST` method, respectively, to the `/tests` URI.

When called with a `GET` it expects an argument `<task_name>` after the URI, specifying the particular task the tests of which are requested, and returns a `200` (ok) status code and a list containing the tests available or a `404` (not found) status code.

When called with a `POST` the request must include a JSON containing the test input and output, as defined by the API. If the information in the body is in the correct form, it stores the new task and returns a response containing a `201` (created) status code. Otherwise it returns a `400` (bad request) status code discarding the request data.

delete_test

This view implements the *Delete Test* operation of the API. It is called using a `DELETE` method to the `/tests/<test_id>` URI. If it exists, the view deletes the specified test and returns a response containing a `204` (no content) status code. Otherwise it returns a `404` (not found status code).

5.2 Back-end Implementation

In this section we describe the way we built the distributed system for our service, based on our design as described in Chapter 4. Then, we continue by presenting the adjustments we made to the old evaluation system to make it compatible with our new architecture.

To create the main components of our distributed system we used ZeroMQ (Section 2.2). These components consist of the broker, the safety system and the workers.

5.2.1 Broker

The broker is an intermediary program that receives submissions from the HTTP servers and the safety service and assigns them to available workers. For that purpose, it uses a pair of ZeroMQ **ROUTER** sockets, one to receive submissions and the other to route these submissions to available workers. It also keeps an internal list of the available workers and another with

the pending submissions. To communicate with the front-end, we use a **ROUTER** socket, because it combines well with the **DEALER** socket we use there, as it can receive consecutive messages without replying back. For the back-end, it is useful to use the ROUTER socket, because we want to route messages to particular workers available.

In more detail, when the broker restarts, it creates the two sockets described and binds them to two endpoints using predefined addresses. Afterwards, it connects to the database and sends an SQL query, asking for submissions with **IN_QUEUE** status. If there are such records in the database, it means that the broker crashed or terminated with pending submissions in its queue. The broker then uses another query to the database to get the relevant task data of these submissions and, with them, it reconstructs the proper submission form. Finally, it pushes the resulting submissions back to its internal queue.

Subsequently, it enters a loop in which it uses a poller to check its queues for incoming messages. If there are messages in the front-end queue, it receives them and tries to dispatch them to available workers. If no workers are available it keeps them in its internal queue.

Any messages in the back-end queue are `READY` messages sent by workers, signifying that they are ready for work. For every work request it receives, it checks its internal queue for submissions and dispatches the first available to the worker from which the request came. This is done by prepending the worker identity, received with the `READY` message, to the front of the submission message. It then connects to the database and updates the submission status to `PROCESSING`, using an appropriate SQL query. If none are available it keeps the worker's identity in the queue of available workers to use it later.

After handling queue traffic, the broker sends a `HEARTBEAT` message to all its idle workers at regular intervals, informing them that it is still running. This is used as part of a mechanism, described later in detail, which ensures that the broker sees all its available workers.

The overall behaviour of the broker is presented in Fig. 5.1

5.2.2 Safety System

This component, as described in Chapter 4, checks the database for submissions that failed to be evaluated completely and sends them back to the front-end broker queue. To do this it uses a **DEALER** ZeroMQ socket, in the same way as the HTTP servers.

When it restarts, it creates the **DEALER** socket and connects to the available broker endpoint. Subsequently, it sends an SQL query to the database asking for all submissions with a **PROCESSING** status, whose timestamp is older than a critical value, which indicates that the evaluation failed, while they were being processed by a worker. If there are such submissions, it uses another query to the database to get the relevant task data of these submissions and incorporates them to the submission data to reconstruct their proper form. Afterwards, it sends all these submissions back to the broker as new evaluation requests. Finally, it updates the status of these submissions to the database to **IN_QUEUE** and their timestamp to current time.

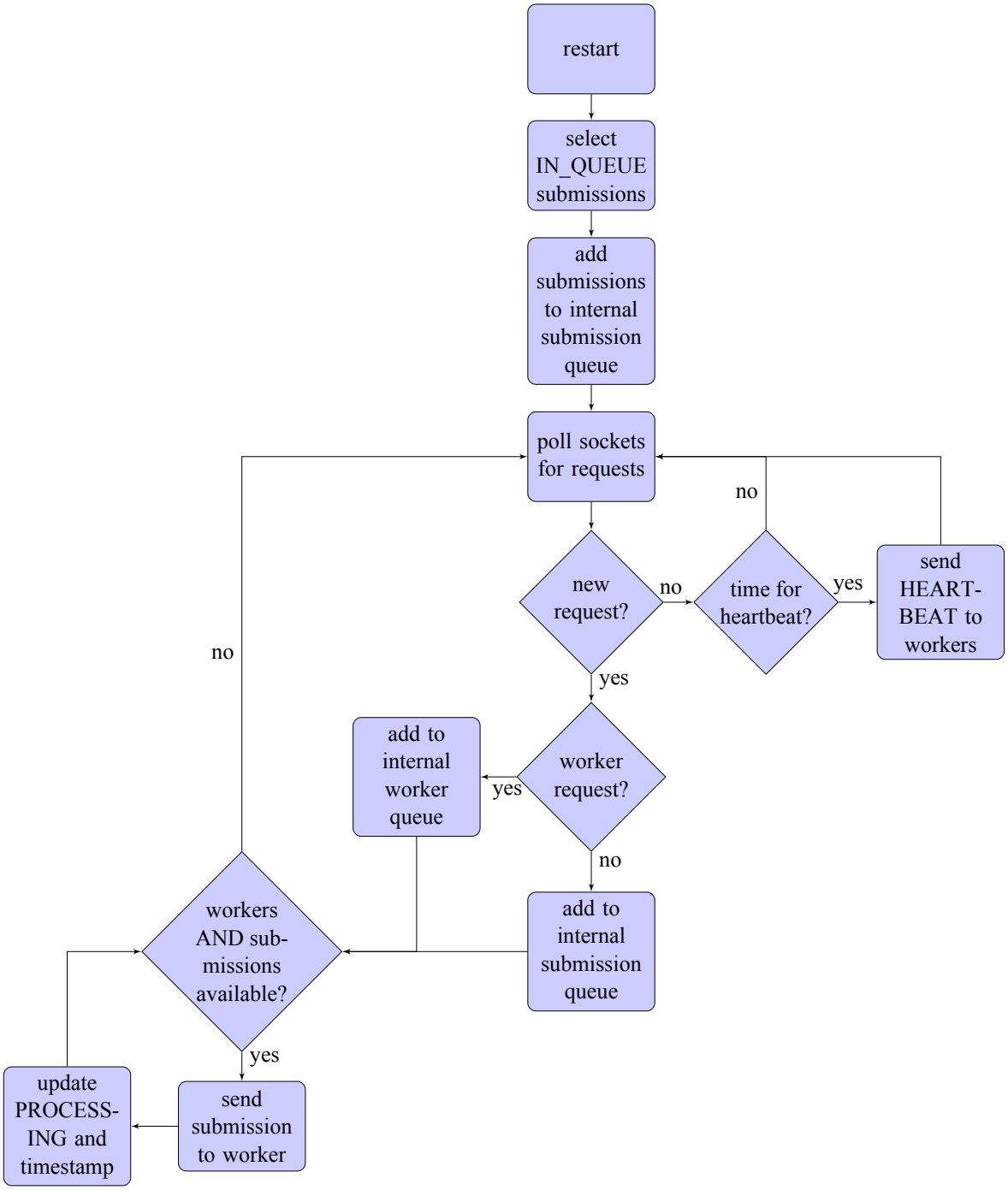

Figure 5.1: The overall behaviour of the broker.

5.2.3 Workers

The workers are the components of the system that evaluate new submissions. For that purpose, they communicate with the broker using **REQ** sockets to ask and receive work and use the older evaluation system to evaluate and grade the submitted programs received. The reason we use **REQ** sockets is because they are synchronous. A worker will send a request and then will expect a reply back.

To start multiple workers on each node, we run a main process which forks, assigns a `worker()` function to each of its children, along with an incrementing integer, as argument, unique for each child, and waits for them indefinitely. The integer is used to distinguish which local files are handled by each of the children. This worker function plays the role of our worker.

Upon being called, each worker process creates a **REQ** socket, connects it to the broker back-end endpoint and sends a `READY` message indicating that it is ready to receive a new submission. To make sure that the broker is online and their `READY` message has reached it, workers use a timer. If they receive a reply from the broker before the timer expires, it means that the broker is still alive. In case no messages are received within that time frame, they cannot be sure if the broker received their request and therefore they have to send it again. As it is not possible to send two consecutive messages with a **REQ** socket, they have to start again, creating a new instance of themselves and send another `READY` message. In addition, in order to provide some time to the failing broker to restart, they wait before doing so, using an exponential backoff strategy. Ultimately, the broker will restart, receive their message, mark them as available, and send a reply back, either in the form of a submission or as a `HEARTBEAT` message.

Without this mechanism, the workers would be unaware about the liveness of the broker. Consequently, in the event of a failure, the broker would restart, but it would not have any available workers in its list, as it would not have received any `READY` messages. Using this technique, we ensure that the workers will persistently ask for a reply, until the broker is back online.

To check for new messages, workers use a poller. As we have explained, there are two types of messages accepted:

- A message containing a new submission to be evaluated. When the worker receives this, it restarts the timer for the broker and proceeds to process the message, by revoking the evaluation mechanism. While evaluating the submission, it stores all the relevant results in a string using JSON format. When the evaluation is completed, it sends this string, using an HTTP `POST` method, to a predefined URL on the HTTP servers. This URL contains the `/submissions/<sub_id>` URI to refer to the particular evaluated submission. It also checks for an optional, user provided callback URL to post the results as well. Afterwards, the worker sends another `READY` message to the broker to indicate that it is available again.

- A `HEARTBEAT` message. When the worker receives this type of message, it restarts the broker timer and continues to poll its queue for requests or other `HEARTBEAT` messages.

5.2.4 Adjustments to the Previous Architecture

In general, we preserved the architecture and the basic components used in the older evaluation system. These include tasks, submissions, tests, and graders as well as the overall behaviour presented in Chapter 3. In this section we describe the changes we made to these components to make the system's architecture compatible to our design.

Tasks

The changes concerning tasks include the following:

- Instead of a special flag field we introduced a "grader" char field, specifying the name of the grader used for the particular task's submissions. A blank grader field indicates default grading.
- Tasks are no longer stored in local files of the evaluation system, but only in the database. The necessary task information for each submitted solution is incorporated in the final form of the submission received by the evaluation system. This information is retrieved from the database and added by the front-end or other intermediaries and is not stored locally.

Submissions

Submissions are also not stored in local files, only in the database, and the queue is now provided through ZeroMQ sockets, from which the system receives the submissions and handles them using appropriate data structures.

Storing submissions in the database is not necessary for the evaluation process, since all the relevant information is sent to workers inside ZeroMQ messages. However, this method is crucial for fault tolerance, as it enables the system to retrieve and fully reconstruct submissions that were lost from the broker or the workers because of a failure.

To compile and execute the source code of the submissions we use special directories `/tmp0, /tmp1, …, /tmp<max_child_id>`, where we use the unique integer assigned to each of the children processes, running evaluation systems, as index. Each child process uses the corresponding directory to save the source code of the submission, in a `<sub_id>.<lang_ext>` file, and its resulting executable, after its compilation. After the evaluation the contents of the directory are erased, so that the contents of a new submission can be stored.

Submissions received for evaluation contain the following fields in our implementation:

- *task_name*: The relevant name of the task.
- *id*: The unique id assigned to each submission by the database.
- *tests*: A list containing the ids of the test cases for this task.
- *code*: The base64 encoded source code of the solution provided.

- *language*: The programming language used for the source code.
- *callback_URL*: An optional, user provided callback URL to send the results.
- *input_file*: The name of the input file for submissions of this task. Should be blank for stdin.
- *output_file*: The name of the output file for submissions of this task. Should be blank for stdout.
- *time_limit*: The time limit for submissions of this task.
- *memory_limit*: The memory limit for submissions of this task.
- *grader*: The name of the grader to be used for submissions of this task. Should be blank for the default grader.
- *g_code*: The source code of the custom grader, if one is specified. Otherwise, this field should be left blank.
- *g_lang*: The programming language used for the grader source code.
- *type*: The type of the relevant task, which can be batch, reactive or output only. The functionality is not yet implemented.

To store all these data we use an appropriate data structure internally, `submission_t`. It is defined as:

```
typedef struct {
    char * task_name;
    int sub_id;
    int test_nbr;
    int * tests;
    double runtime;
    char * code;
    char * language;
    char * callback_url;
    char * input_file;
    char * output_file;
    double time_limit;
    int memory_limit;
    char * grader;
    char * g_code;
    char * g_lang;
    int type;
} submission_t;
```

Listing 5.1: The `submission_t` data structure we used for submissions.

Tests

To keep a consistent view of the test case data among the workers we use the NFS protocol [1]. Specifically, we store the test case data as files in a separate node which is also an NFS server. This node then shares the data to the workers, where each worker in this case also acts as an NFS client. For each submission, the appropriate test case data are copied to the corresponding `/tmp<child_id>` directory in the texttt<task_name>.in file which after the completion of the evaluation is replaced by the data of the next test case.

Graders

Graders, like the rest of the components described, are not stored locally in the file system of the worker. Their data exist only in the database and are incorporated to the submission message before it is issued to the broker's queue. As is the case with submissions, graders, apart from the default grader, need also to be compiled and executed. Therefore, their code is stored in a temporary file `grader.<lang_ext>`, under the appropriate `/tmp<child_id>` directory.

The default grader is an internal function which is called instead of executed. Therefore we do not store it in any particular files.

In contrast with the existing evaluation system, where each task had a dedicated grader, either the default or a custom one, we now provide a list of graders available to all tasks, with each submission specifying the desired grader to be used.

5.2.5 Evaluation System Behaviour

The overall behaviour of the system is similar to the behaviour presented in Chapter 3. The changes we made are only aimed to incorporate our new distributed architecture. The resulting behaviour is presented in Fig. 5.2 and can generally be described in the following steps:

1. Receive a new request for evaluation from the worker socket.
2. Store the information included to a `submission_t` struct. All source code included is decoded before being stored.
3. Store the source code of the solution provided in a file `<sub_id>.<lang_ext>` under the appropriate `/tmp<child_id>` directory.
4. Compile the source code. The results (successful compilation/errors) are stored in a string.
5. After successful compilation, check if a custom grader is specified. In that case copy the source code of the grader to a `grader.<lang_ext>` file under the same `/tmp<child_id>` directory and compile it, storing the results to a string. If any of the aforementioned compilations fails jump to step 7.

6. `i=1;`
   While `i <= test_cnt`:
   - Check which test case id is specified in the `tests[i]` variable and copy its data to the `/tmp<child_id>` directory, in the appropriate file, which is named `<input_file>.in`.
   - Run the executable of the solution submitted.
   - Check which grader is specified and either call the appropriate function or run its executable.
   - Collect the results and store them in an appropriately formatted string.
   - `i++;`
7. Send the results collected to the HTTP servers and, if provided, to the user defined callback URL.
8. Send a `READY` message to the broker.
9. Poll the queue for incoming messages.

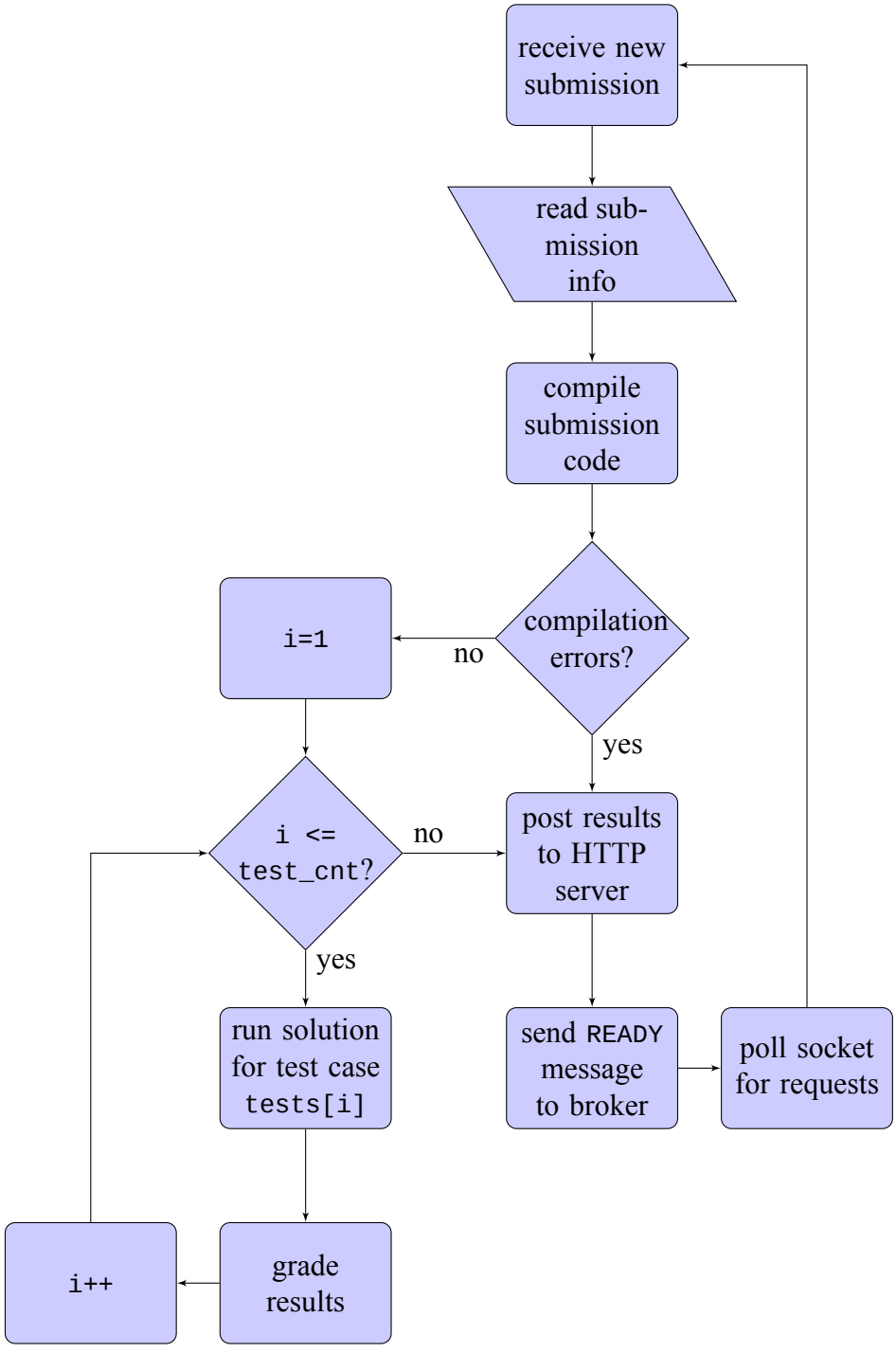

Figure 5.2: The overall behaviour of the adjusted evaluation system.

Chapter 6

Benchmarks

6.1 Testbed description

To test our system we used a computer node with four Intel Xeon® E7340 (2.40 GHz) processors, with a total number of 16 cores, 4 MB cache and 16 GB RAM, and Debian GNU/Linux 8.0 operating system.

6.2 Evaluation of the new architecture

In this section, we evaluate the performance of our distributed architecture. In order to achieve this we deploy our system on the testbed node, submit multiple solutions for a simple task and measure the time needed to evaluate them using different numbers of workers. The submissions are multiple copies of the same implementation of a `O(NlogN)` algorithm, which we execute against two test cases, one for N = 1000 and another for N = 20000. The results are shown in Fig. 6.1 and Fig. 6.2 (execution time and speedup, respectively).

We notice that our architecture provides initially an almost linear increase in performance, which is expected, as each evaluation of a submission is an independent task, conducted entirely by one worker. As a result there is no need for communication between workers, which means that the speedup achieved in the evaluation process should be proportionate to the number of workers available. One limitation in the number of possible workers is imposed by the physical characteristics of the system. As the number of workers approaches the number of physical cores available, the speedup becomes significantly less than linear. This is a standard overhead resulting from the fact that there are other processes running on the system, such as the broker, the HTTP server and the safety service. Another standard overhead derives from the message passing among the components of our system.

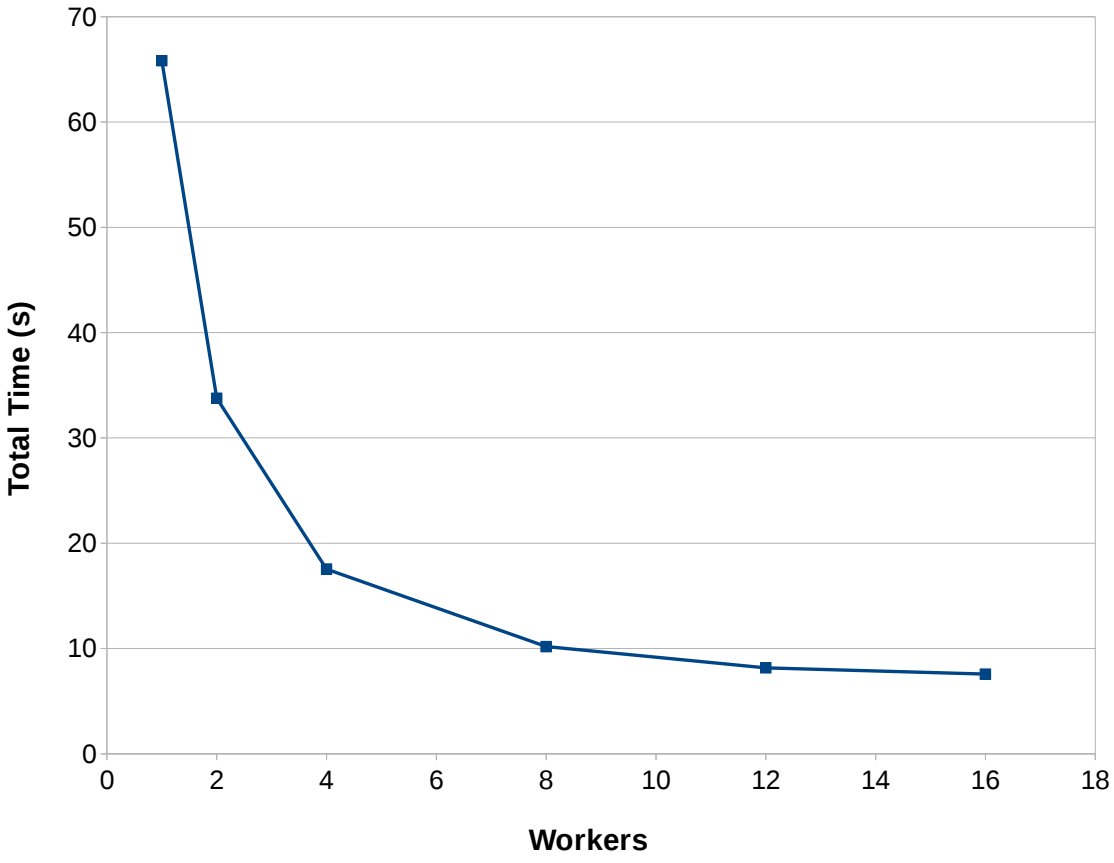

Figure 6.1: The time needed to evaluate 200 submissions

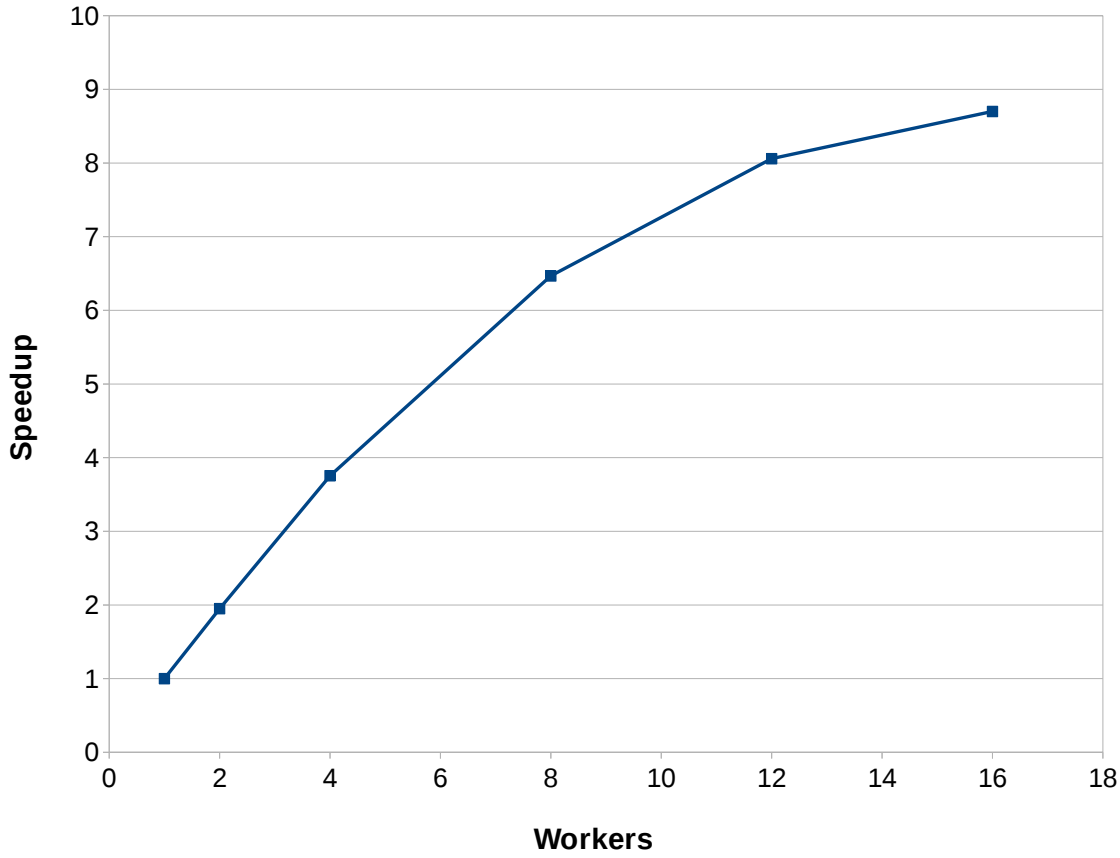

Figure 6.2: The speedup achieved for 200 submissions

Chapter 7

Conclusion

7.1 Concluding remarks

This thesis is an effort to increase the performance and availability of the automatic evaluation system by designing and implementing a distributed architecture for it. It also deals with the issue of designing and implementing a RESTful API to provide the grading capabilities as a service.

The performance evaluation we conducted shows that our architecture successfully provides the system with these desired properties and, consequently, is a much better alternative to the original architecture of the system.

In addition, our API provides a unified interface to access the grading services publicly and it is therefore an efficient way for interested clients to make use of our service.

7.2 Future work

We intend to deploy our proposed architecture on a cluster of our university division and use it for class assignments and competitions. We also need to further expand the system to provide grading for more kinds of tasks, like reactive tasks, which is currently not supported, or to increase support for custom grading tasks. Similarly, we can add grading support for more programming languages.

Furthermore, we intend to increase system security by creating a better sandboxed environment to test the submissions, since student programs cannot be trusted [8]. We would also like to provide our service with elasticity to automatically adjust the resources according to the workload.

Moreover, since there are very few writes and many reads on the NFS server, we can develop a caching mechanism for test cases to increase performance. Finally, we can develop a test case specification language so that the grader can automatically check that the format of the test cases complies with the specification provided in the task description.

Bibliography

[1] Brent Callaghan, Brian Pawlowski, and Peter Staubach. NFS version 3 protocol specification. Network Working Group, 1995.

[2] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. Distributed Systems: Concepts and Design. Pearson Education, Inc., 2012.

[3] Thomas Erl. SOA: Principles of Service Design, volume 1. Upper Saddle River: Prentice Hall, 2008.

[4] Filho Ferreira and Freitas Otávio. Semantic Web Services: A RESTful Approach. IADIS, 2009.

[5] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. Dissertation, University of California, 2000.

[6] Sukumar Ghosh. Distributed Systems – An Algorithmic Approach. Chapman & Hall, CRC, 2007.

[7] Pieter Hintjens. ZeroMQ: Messaging for many applications. O'Reilly Media, Inc., 2013.

[8] Martin Mareš and Bernard Blackham. A New Contest Sandbox, volume 6 of Olympiads in Informatics. Vilnius University, 2012.