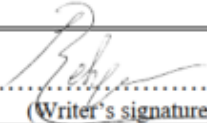




University of
Stavanger

Faculty of Science and Technology

MASTER'S THESIS

Study program/ Specialization: Computer Science - Reliable and Secure Systems	Spring semester, 2020. Open / Restricted access
Writer: Behfar Behzad	 (Writer's signature)
Faculty supervisor: External supervisor(s):	Antorweep Chakravorty and Dhanya Therese Jose
Thesis title:	Implementation of TOTEM defined SDK
Credits (ECTS): 30	
Key words: Blockchain, Big Data System, Hyperledger Fabric, Apache Hadoop, SDK	Pages:42..... + enclosure: Appendix A Appendix B Stavanger, ...7/7/2020..... Date/year



UNIVERSITY OF STAVANGER

Implementation of TOTEM defined SDK

Author:

Behfar Behzad

Supervisors:

Antorweep Chakravorty

Dhanya Therese Jose

July 7, 2020

Abstract

Blockchain Technology made its first debut when Satoshi Nakamoto, whose real identity is still unknown, released the white paper *Bitcoin: A Peer to Peer Electronic Cash System* in 2008. This technology has developed over the last decade with the primary mission to establish a creditworthy distributed environment among various independent contributors in a non-trustable manner. The transparency, tamper-proof records, and decentralized nature of blockchain have grabbed the attention of many scientists and investors to solve old business problems in new ways.

At the same time, with Internet usage surging over the past decade, a variety of enormous data has been generated at a breakneck pace from various sources, whether it is from social media, banking sectors, governments, etc. As a result, many organizations throughout the globe have changed their work culture and adopted Big Data analytics to gain various benefits from the data being produced. Nevertheless, handling these big data was always a tremendous challenge for scientists and engineers as it involved large and complex information, which cannot be handled by conventional tools.

TOTEM: Token for controlled computation, accounts for a new concept that integrates both blockchain technologies and big data systems and uses their advantages to present a better, safer, and more cost-effective solution for both data consumers and providers. The TOTEM project's main objective is to overcome the security and privacy breaches and prevent moving large data sets across the network for analysis.

Within the TOTEM computational system, TOTEM SDK is of importance as data consumers use it to create a MapReduce code for computation, which will be performed in the data provider's environment. Depending on consumers' computational needs, pre-defined totems will be assigned to these authorized users. The SDK, along with a smart contract, forms a monitoring system that keeps track of totem value associated with users' submitted codes using an estimator table to allow codes for execution. That is how the TOTEM system puts constraints on computational operations.

This thesis will focus on specific aspects of the TOTEM project, looking into how the SDK interacts with the input data and the other components, takes the code and analyzes it within its layers, and finally, how responds to it. With SDK as the gate to connect the data consumer to the owner of data, it ensures the accountability and transparency of the results based on the rules and language defined for it. How is the SDK developed and implemented? What is the architecture used in it? These are the questions that the present study will look into in further detail.

Acknowledgements

I would like to express my gratitude and appreciation to my supervisors, Assoc. Prof. Antorweep Chakravorty and Mrs. Dhanya Therese Jose, for their valuable advises and feedback throughout my work on this thesis.

I am also extremely indebted and thankful to my family, especially my lovely wife, for so much affection, care and blessings.

Contents

1	Introduction	1
1.1	Contributions and Outline	2
2	Background	3
2.1	Blockchain Technology	3
2.1.1	Smart Contracts	4
2.1.2	Hyperledger Fabric	6
2.2	Big Data System	7
2.2.1	Hadoop	8
	Hadoop Distributed File System	8
	MapReduce	9
2.3	JavaScript	10
2.3.1	Node.js	10
2.4	Related Works	11
3	Architecture	13
3.1	Theory of TOTEM	13
3.2	The workflow in the TOTEM architecture	16
3.2.1	Data Consumers Side	16
	Totem Estimator Table	17
3.2.2	Data Providers Side	18
	Customized Computational Framework	20
3.3	Architecture of TOTEM defined SDK	22
3.3.1	Specifications	22
3.3.2	Layers	23
	SDK layer	24
	Controllers layer	25
	Handlers layer	25
4	Implementation and Results	29
4.1	Frameworks and Tools	29
4.2	Coding Explication	32
4.2.1	Environment Setup	32

4.2.2	Main Functionalities	34
4.3	Execution and Testing	37
5	Conclusion and Future Work	41
	Bibliography	47
A	TOTEM SDK User Manual	52
B	Configuration guide	54

List of Abbreviations

AJAX	Asynchronous JavaScript And XML
CA	Certificate Authority
CCF	Customized Computational Framework
CLI	Command Line Interface
COR	Cross Origin Requests
CORS	Cross-Origin Resource Sharing
CSA	Cross Site Attacks
CSC	Control Statement Controller
DFS	Distributed File System
DLT	Distributed Ledger Technology
DN	Data Node
DNS	Domain Name System
DOM	Document Object Model
DoS	Denial of Service
EHC	Error Handler Component
ET	Estimated Totem
ETH	Ethers (cryptocurrency of Ethereum network)
GFS	Google File System
GUI	Graphical User Interface
HDFS	Hadoop Distributed File System
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure

I/O	Input and Output
IBAC	Identity-Based Access Control
IoT	Internet of Things
JS	JavaScript
LRC	Line Reader Component
LTC	Labelled Type Controller
MDP	Modular Design Pattern
MSP	Membership Service Provider
NPM	Node Package Manager
OHC	Operation Error Handler
OSN	Ordering Service Nodes
P2P	Peer To Peer
PoS	Proof of Stake
PoW	Proof of Work
RBAC	Role-Based Access Control
RC	Reaction Checker
RDD	Resilient Distributed Dataset
RPC	Rule Processor Component
SDK	Software Development Kit
SHDFS	Simplified Hadoop Distributed File System
TCP	Transmission Control Protocol
TDA	Trusted Distributed Application
TET	Totem Estimator Table
TL	Totem Language
TOTEM	Token for controlled computation
UDP	User Datagram Protocol
UI	User EInterface
URL	Uniform Resource Locator

UX **User Experience**
VC **Variable Checker**

Dedicated to my parents, in whose hands I had received my real education, and to my wife, Maryam, for giving me strength to reach for the stars and chase my dreams.

Chapter 1

Introduction

This thesis will focus on the implementation of a Software Development Kit (SDK) for a novel architecture integrating the Hyperledger Fabric blockchain and the Apache Hadoop framework, investigating its architectural design process and the methodologies employed to develop the SDK using JavaScript. The thesis will also look into the tools, frameworks, and the chief functionalities used during the development of the SDK current version, for the purpose of policies and coding procedures.

In today's era, with the role of technology becoming multi-fold in all sectors, we need to deal with a boom in the data industry that keeps increasing exponentially. Nevertheless, conventional methods for data analysis are no more effective as they demand moving data across the network for analysis. Additionally, data owners are also reluctant to expose their data due to security issues. As a result, having infrastructures that are configured and managed to handle the storage mechanism while fulfilling the security and privacy aspects are essential. TOTEM is a new approach that introduces moving the computational code to the data without any concern regarding the data disclosure to external networks. TOTEM employs the properties of Hadoop framework, for its parallel computation on large datasets, and Blockchain technology, for its guarantee on secured and tamper-proof transactions, to enable a new way of computation upon large datasets securely and effectively.

As the aim of this work is the implementation and defining of a TOTEM defined SDK, a natural aspect of developing a set of tools that need to work synchronously with other components would be to investigate what are the comprising components of TOTEM. Overall, the primary objectives of the present study can be divided into two parts. First is to understand the significant components of the TOTEM architecture and the various steps within the workflow of the proposed architecture to study relevant parameters that affect the way that the SDK operates. Secondly, a multi-layer architecture is proposed, and a detailed explanation of the theory and practice of the SDK implementation is provided.

1.1 Contributions and Outline

The following contributions are made in this thesis:

- Designed a 3-tier architectural TOTEM defined SDK, with each layer corresponding to a different service or integration.
- Developed the TOTEM SDK adapted to the Modular Design Pattern (MDP) using Node Package Manager (NPM) packages installed for easy improvement, update, and reusability.
- Defined the TOTEM Language (TL), allowing specific data types and operations to perform in the SDK.
- Implemented the rules in a tabular format as a comma-separated values file with respective rules that TL must follow.
- Implemented the TOTEM defined SDK using pure JavaScript code.
- Developed a Graphical User Interface (GUI), resembling the environment where Data Consumers can submit their computational codes and receive feedback.
- Evaluated the performance of the TOTEM SDK under two testing scenarios.

The remainder of this thesis is outlined as follows:

Chapter 2: Gives an overview of two leading technologies employed in TOTEM: the Blockchain technology and Big Data systems, looking at Hadoop as one of the leading frameworks for processing of large data sets and the key categories in blockchain; permissionless and permissioned.

Chapter 3: Elaborates on the structural design in the TOTEM and its defined SDK, studying the interactions among the TOTEM consisting components and proposing a multi-layered architecture for the SDK.

Chapter 4: Presents the methods, tools, and foundational functionalities implemented in the SDK and evaluate tests conducted over two defined scenarios.

Chapter 5: Concludes the work of this thesis and presents ideas for future work.

Chapter 2

Background

Blockchain and Big Data are among the leading-edge technologies which have grabbed a great deal of attention. Both are expected to reshape the way businesses are done across all kinds of industries in the years ahead. After a long time of developing in full swing, one might assume that Blockchain and Big Data are quiet two orthogonal technologies. However, the idea is rapidly changing. The Distributed Ledger Technology (DLT) that powers Bitcoin, Ethereum, and other cryptocurrencies is expected to help enterprises to get to grips with Big Data finally. This technology can revolutionize conventional methods for data analysis, which has presented a slew of problems, such as dirty data, inaccessible data, and security breaches. Token for controlled computation (TOTEM), which has just been introduced by computer scientists at the University of Stavanger in [1], rose to these challenges by combining the Blockchain and Big Data system.

This chapter will focus on Blockchain Technology and its primary types, as well as the design and architecture in the Big Data System. A presentation of Hadoop and its core functionalities will be followed by a short comparison between two popular Big Data frameworks. A detailed consideration into the structural layers will also be illustrated, to provide a more straightforward understanding of the concept in TOTEM defined SDK presented in the following chapters.

After covering Blockchain Technology and Big Data System, attention will shift to the JavaScript as the programming language by which the defined SDK is implemented. Node.js, as the server-side runtime environment for JavaScript, will also be presented and discussed, referencing related works from other parties.

2.1 Blockchain Technology

Blockchain is an open distributed ledger that allows multiple parties to add and store transactions into it efficiently, verifiably, and permanently. With Blockchain, data can

be embedded in digital code and be recorded in transparent, shared databases. This method makes Blockchain protected against revision, tampering, and deletion [2]. It is, in fact, a Point to Point (P2P) communication among all nodes in the chain; therefore, no central node can monitor or control the whole network. In [3], A. Meier introduces Blockchain as a combination of distributed ledger and consensus algorithm in which the participating nodes of this P2P system guarantees the protection and integrity globally, thanks to the consensus algorithm.

Cryptographically hash functions are implemented to record the data into the blockchain. In 1991, S. Haber and W. Stornetta described cryptographically secured chains of blocks to make the document timestamps tamper-proof [4], and one year later, in 1992, they used Merkle tree to allow timestamping of several documents into one block. After 17 years of attention on this concept in 2009, Bitcoin was first introduced by Satoshi Nakamoto [5]. Bitcoin is an entirely decentralized, permissionless P2P Blockchain.

Blockchains are classified into two types of public and private blockchains [6]. The public Blockchain (a.k.a. Permissionless Blockchain) is permissionless, signifying that contributors are anonymous, and no restriction in joining the network or authentication process is required. Bitcoin and Ethereum are significant examples of public blockchains. On the other hand, in the private Blockchain (a.k.a. Permissioned Blockchain), prior approval is necessary, and participants are restricted to get permission to join the network. Hyperledger Fabric ¹ and Ripple ² are two examples of private blockchains.

Due to the decentralized nature of blockchains, the validity of each proposed transaction can be verified by any node in the network. These nodes add transactions to a block and append it to the existing chain. However, some nodes may come up with a new block to append simultaneously. Thus, there must be an agreement among distributed nodes about which node can append a new block. This is where the consensus algorithm comes to play to guarantee that all rules are being followed in a trustless way. Consensus algorithms can be proof-based, such as Proof of Work (PoW) and Proof of Stake (PoS), or voting based [7].

2.1.1 Smart Contracts

Smart contracts account for one of the promising uses of Blockchain that has exploited this technology's potential even beyond cryptocurrencies. In [8], Nick Szabo, cryptographer and lawyer, was the first person who coined the term in the 1990s. He suggested to translate the contract terms into computer codes and added them into the software or hardware. This suggestion not only makes the contract automated and self-executed, but it also minimizes the possibility of accidental exceptions and

¹<https://www.hyperledger.org/>

²<https://ripple.com/>

fraudulent transactions between parties [9]. Since then, people have defined smart contracts in different ways. C. D. Clack in [10] provides a broad definition of *smart contracts* that integrates all various uses:

“A smart contract is an automobile and enforceable agreement. Automatable by computer, although some parts may require human input and control. Enforceable either by legal enforcement of rights and obligations or via tamper-proof execution of computer code.”

This low-level computer code deployed on blockchains (without trusting a third-party) monitors all transactions so that each transaction in the Blockchain is executed if they meet all the terms and requirements in the smart contracts. A Smart contract operates as a Trusted Distributed Application (TDA), which has obtained its security from the Blockchain and the consensus algorithm shared among peers.

In permissionless blockchains, where all nodes join anonymously and participate in validation, each node may deploy a smart contract which requires high computation, particularly in PoW algorithms. If the smart contract requires high execution time to implement, substantial delays might occur in the network, which makes the entire network vulnerable to malicious attacks such as Denial-of-Service (DoS) attack [11]. Consequently, it is necessary to restrict the complexity of smart contracts to prevent additional delays to the network. One of the best solutions to this issue is employed by Ethereum called *gas*. Figure 2.1 from [12] depicts an example of making a transaction in the Ethereum network.

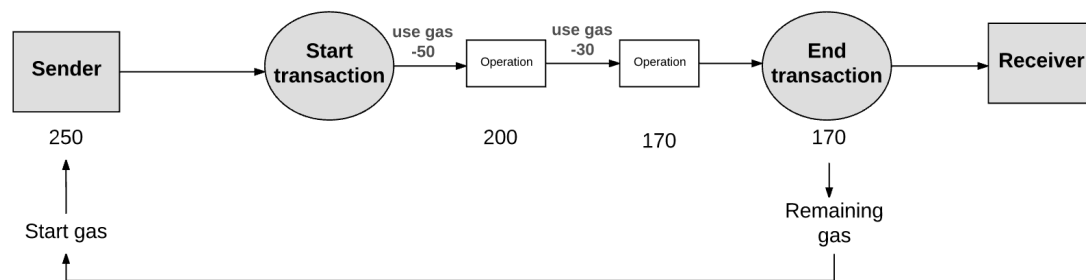


FIGURE 2.1: Ethereum transaction example using gas fee [12]

Gas is an internal pricing mechanism that measures how much does a transaction would cost in computing resources. Nodes must pay the gas fee in Ethers (ETH) to add their transactions to blocks. The more gas they pay, the faster their transactions are mined by miners. A transaction fails if the smart contract exhausts the available gas of the creator node. The gas mechanism is one of the successful approaches to allocate resources efficiently and reduce unrealistic spam on the network [9].

2.1.2 Hyperledger Fabric

Fabric is a module-based architecture and extensible open-source foundation to establish, manage, and operate on private or permissioned blockchains [13]. In Hyperledger Fabric, each contributor should register to the network through a trusted Membership Service Provider (MSP). The MSP maintains the information about all nodes in the network, including peers, clients, and Ordering-Service Nodes (OSN), and its primary role is to issue credentials used in the authentication process. The Hyperledger fabric allows different MSPs, and there is no restriction in which format the ledger data is stored.

Ledger is a critical concept in Hyperledger Fabric. It contains factual information about objects, both the current value and attributes and the transaction history that led to these values [13, 14]. A ledger in Hyperledger Fabric consists of two different but related components – world state and transaction logs (Blockchain). World state is implemented like a database and holds current values and attributes of an object, while transaction logs (Blockchain) is the updated historical record for the world state [15].

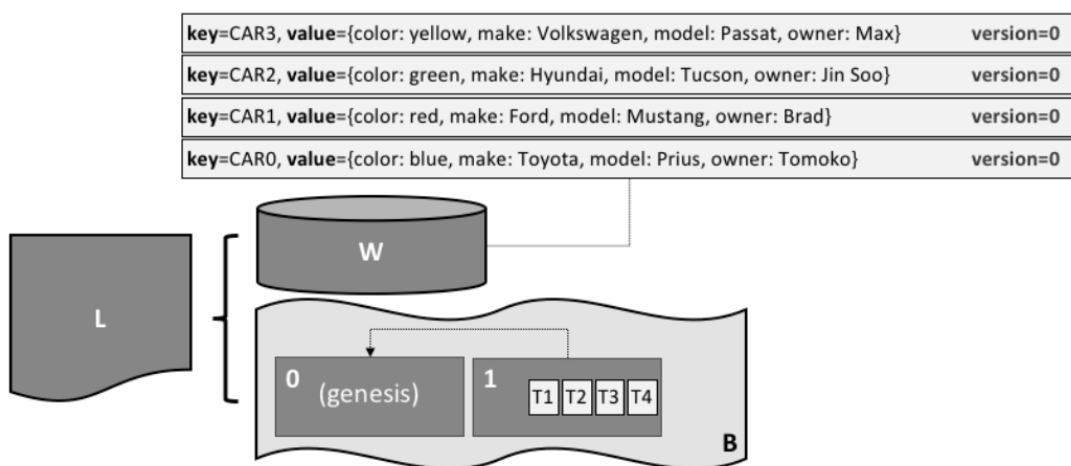


FIGURE 2.2: Example Ledger: fabcar [15]

Figure 2.2 indicates a sample ledger L comprises of a world state W and transaction log (Blockchain) B. W has four states with keys CAR0 to CAR3. All states are at version 0. B consists of two blocks: the genesis block (first block of a blockchain) and block 1 with four transaction T1 to T4 [15].

One of the other advantages of using Hyperledger Fabric is to run arbitrary smart contracts in them. Smart contracts in Fabric, called *chaincode*, are usually implemented in Golang³ or Node.js⁴ programming languages; the chaincode is installed on each peer in the network and ready to be invoked by applications which require to interact with the ledger [14].

³<https://golang.org/>

⁴<https://nodejs.org/en/>

TABLE 2.1: A comparison between two popular big data frameworks

	Hadoop	Spark
Data format	Key-Value	Key-Value, RDD
Processing mode	Batch	Batch and Stream
Programming model	MapReduce	Transformation and Action
Supported programming language	Java	Java-Scala-Python
Iterative Computation	Yes	Yes

2.2 Big Data System

Big data accounts for large-scale datasets defined as the combination of large, unstructured, complex, and heterogeneous datasets. These are beyond the capabilities of standard analytical methods in data management such as process, store, capture, analyze, and visualize in an acceptable amount of time [16]. Big data is usually characterized by four Vs – volume, velocity, variety, and veracity. These Vs symbolize the size of data that is stored, the rate at which data is captured and processed, various types of data stored and how much the data is accurate [17].

There are various methods, tools, or strategies developed to analyze significant volumes of data. Among all, some are more popular due to different reasons such as iterative and interactive analytics, lower memory consumption, disk bandwidth improvement, etc. [17]. Hadoop ⁵, Spark ⁶, Storm ⁷, Samza ⁸, MongoDB ⁹ and Cassandra ¹⁰ are popular examples of such tools and frameworks.

Table 2.1 illustrates some of the key differences between Hadoop and Spark frameworks as the most popular tools in Big Data analytics. This comparison is carried out according to the data format, processing mode, programming model, fault tolerance, and whether the framework allows iterative computation [18].

Regarding data formats, in key-value type keys and values are encoded as tuples, which makes the Read/Write operations so fast. At the same time, Resilient Distributed Datasets (RDD) is a read-only partitioned collection of records which reduces speed but gives immutability to datasets. Batch Processing mode also processes over all or most of the data, while Stream Processing only processes the most recent records of data. This is the reason why Spark is approximately 100 times faster in processing than Hadoop. Finally, Hadoop uses MapReduce programming model, which gives it the advantage of processing data sets parallelly. This model makes Hadoop work on more extensive data sets than Spark does in which Transformation and Action model is employed [16].

⁵<https://hadoop.apache.org/>

⁶<https://spark.apache.org/>

⁷<https://storm.apache.org/>

⁸<http://samza.apache.org/>

⁹<https://www.mongodb.com/>

¹⁰<https://cassandra.apache.org/>

This short comparison makes Hadoop a better option to be employed in TOTEM architecture with large clusters of data. However, all tools and frameworks have their strengths and weaknesses.

2.2.1 Hadoop

Hadoop is an Apache project that began in 2008 and accounts for one of the leading frameworks in distributed processing of big data sets with clusters of computers. Hadoop is an open-source framework that works on a programming model approach, namely MapReduce to process and generate large data sets [19]. Hadoop principally consists of a two-layer structure, designed to improve the performance of handling I/O requests [20]. These layers are called the Hadoop Distributed File System (HDFS) and MapReduce (distributed processing).

Hadoop Distributed File System

HDFS is the implementation of Google File System (GFS) and provides a scalable Distributed File System (DFS) to record big files on distributed machines reliably and effectively. Figure 2.3 from [18], illustrates an overview of HDFS architecture and its components. As it is shown, HDFS has a master/slave architecture where the Name Node is the master with several Data Nodes being slaves.

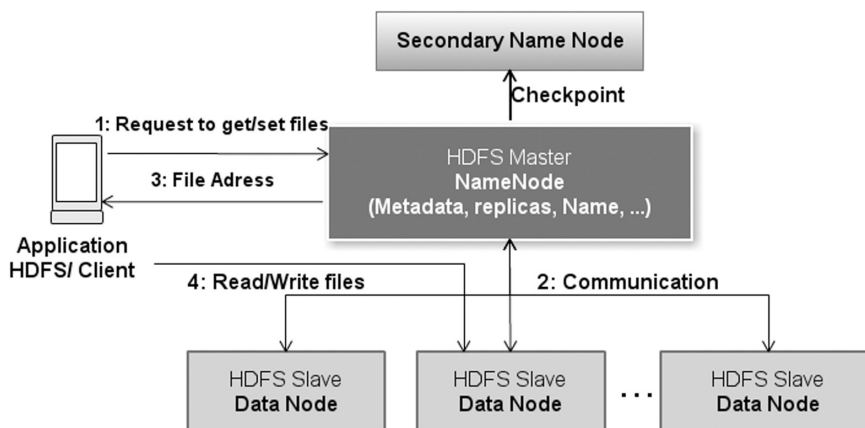


FIGURE 2.3: An overview of HDFS architecture [18]

The Name Node or the master is mainly responsible for providing physical space to record massive data files sent by the HDFS client. Data Nodes or slaves, on the other hand, are used to store HDFS client data files after files being split into fixed-sized blocks. The Name Node determines how the data blocks should be mapped into the slaves, and slaves are responsible for doing the Read/Write operations such as creation, deletion, or replication of data blocks. Data Nodes also send periodic heartbeat signals to the master to indicate they are active. If the HDFS client wants to retrieve data from HDFS, it sends a request to the master. The master, then, will seek for the data location in its indexing system and sends the address back to the HDFS

client as file system metadata (file name, file location, etc.). There is also a secondary Name Node, which stores periodically the leading Name Node state and checkpoints of the metadata to play the same role if the main master fails [18].

MapReduce

MapReduce is a programming model for processing and producing large data sets. This model is built in 2003 to achieve a simplified way of constructing an inverted index in order to handle searches at Google.com. MapReduce is a highly efficient framework for large-scale data analysis [19]. MapReduce distributed processing framework consists of two principle functions or tasks that are executed: map and reduce. Users specify a map function that takes a key-value pair as input and generates a set of intermediate key-value pairs as output [21]. The following script from [19] shows pseudo-code for map function that a user would write:

SCRIPT 2.1: Map function in pseudo-code

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    DeleteIntermediate(w, 1 );
```

In the reduce function, the intermediate key-value pairs (map function's outputs) are merged and grouped by key. For each group, the user performs the reduce function, producing new key-value tuples associated with a unique key [21]. All the data, including inputs and outputs of both functions, will be stored on the file system. A user would write the reduce function like the following pseudo-code from [19]:

SCRIPT 2.2: Reduce function in pseudo-code

```
reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Delete(AsString(result));
```

MapReduce framework is based on two components: Job Trackers or masters that manage and control resources and schedules of the Task Trackers or slaves. Task Trackers or slaves that execute the given tasks supervised by the Job Trackers, which are mainly supervising the Map/Reduce functions [18]. Slaves send their state data to the masters regularly, and if one fails, The Job Tracker will reschedule the unfinished tasks to the next available Task Tracker. However, if the master fails, the whole system will go down.

2.3 JavaScript

JavaScript (JS) is an imperative, object-oriented language first announced in 1995 by Netscape as an “easy-to-use object scripting language designed for creating live on-line applications that link together objects and resources on both clients and servers” [22]. Since then, JavaScript has become the standard for front-end scripting and even one of the most dominant languages in the software industry. Unlike other popular traditional languages such as Java and C, JavaScript does not allow encapsulation by using classes or structured programming to maximize flexibility. JavaScript prosperity is undeniable to the extent that, based on Google’s report as a data point, it has been used in 97 out of 100 most popular websites and web applications [23].

Often, developers fall in love with JavaScript because of its extremely dynamic nature, including libraries that can be downloaded at run time from various sources on the web, script, and objects in particular. In JS, objects can be sent over the web as raw strings that can be dynamically parsed and easily executed by the receivers, and its APIs support by all modern browsers [24]. Here are two of the distinct advantages of using JS in software or application development over the other programming languages:

- **Less load on sever** – User input can be validated before sending the data off to the server. This action not only saves part of the traffic and demand on the server, but it also increases the interactivity of the software.
- **Speed** – Since JS can be run immediately on the user’s device instead of the server, it reduces server requests and improves the User Experience (UX).

2.3.1 Node.js

Node.js – also called Node – is an open-source, server-side JavaScript runtime environment that runs JS scripts outside of the web browser. The runtime environment compiles JS using Google’s V8 JS engine, and its core is implemented in a JavaScript library. Node simplifies the creation of multi-functional web servers as well as networking tools by using JavaScript. It provides a collection of modules such as file system I/O, data streams, networking protocols, including HyperText Transfer Protocol (HTTP), Domain Name System (DNS), and Transmission Control Protocol (TCP)/User Datagram Protocol (UDP), etc. [25].

Node.js is primarily used to bring event-driven programming to web servers, owing to make the server development fast in JS. It allows developers to build scalable servers without the use of threading as it used to be traditionally, but by using an event-driven programming model that uses call-back functions to indicate that a task is completed [25].

Node Package Manager (NPM ¹¹) is a free to use, software package manager and installer for Node libraries and applications. It includes a Command Line Interface (CLI), used to automate the package management and dependencies inside a file, called *package.json*, with a single line of code.

2.4 Related Works

In “Blockchain solutions for Big Data challenges” [26], the author reviewed the possibilities of employing Blockchain on Big Data systems to alter the currently used model of collecting and managing data in organizations. The potential improvement of these possible solutions was also discussed on various subjects and scenarios, such as decentralized control of personal data, big data in Healthcare, Internet of Things (IoT) communications, and intellectual property of digital art.

In [27], the Simplified HDFS (SHDFS) architecture employs the KERBROS authentication mechanism to overcome the existing disadvantages of HDFS. In the proposed architecture, the Name Node concept is eliminated due to the dependency of the entire system on the Name Node, which leads to the single point of failure problem. Instead, a collection of Data Nodes (DN) called HDFS_DNs is implemented as a cluster, that also makes the design cheaper and more straightforward.

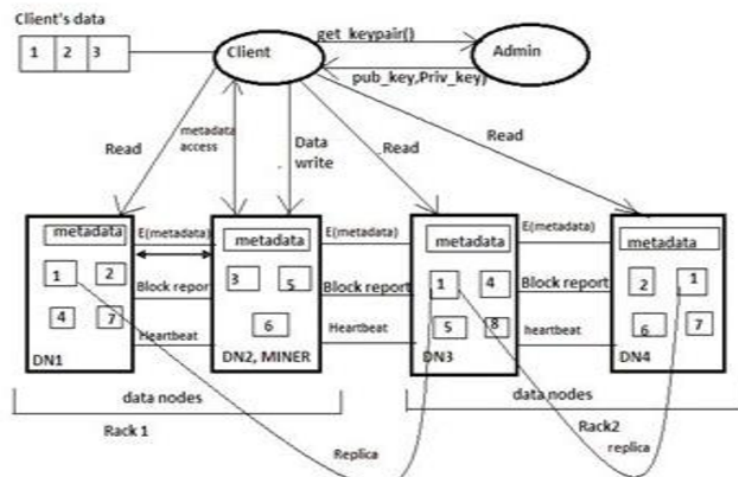


FIGURE 2.4: The SHDFS architecture [27]

As it is depicted in Figure 2.4, the metadata storage is also distributed across all the DNs by a master node called SHDFS_MINER. Blockchain technology is used to provide easy access to linking the data blocks in different Data Nodes serially.

¹¹<https://www.npmjs.com/>

HBasechainDB [28], is a scalable Blockchain-based tamper-proofed Big Data storage for distributed computing. It adds the characteristic values of Blockchain, such as decentralization and immutability, to the Hadoop HBase database. Authors introduced HBasechainDB as a distributed, inviolable, decentralized Big Datastore and considered as a convenient option for enterprises/organizations whose domain logic is built on Hadoop and are willing to adapt Blockchain Technology.

A prototype of the Blockchain Access Control Ecosystem, which provides efficient management in access control of large data sets and guarantees against security breaches and violations for data authorities, is proposed in the “Blockchain access control Ecosystem for Big Data security” [29]. The described architecture designed with the assumption that it will be running in a partial trust environment. The proposed prototype uses a decentralized security system based on Hyperledger Fabric. The Hyperledger Fabric blockchain, as well as two existing access control paradigms – Identity-Based Access Control (IBAC) and Role-Based Access Control (RBAC) – are used to implement the access control of big data. The blockchain technology, in fact, solves the challenges associated with centralized access control and ensures the verifiability, security, and traceability of the data for the authorized owner.

Chapter 3

Architecture

The TOTEM defined SDK in [1] is planned to enable the data consumers to submit their computational codes within the data owner's environment. Furthermore, SDK is responsible for estimating and checking the required TOTEM value that any computational code might cost. If the prerequisites are met in this computational system, SDK will create a MapReduce formatted code for computation. The upshot of all this is that TOTEM SDK plays a vital role in the TOTEM architecture so that it not only must prevent the execution of malicious functions by throwing error, but it also makes the computation ready for the execution part. Hence, a well-structured multi-layer architecture is proposed to handle all the necessary functionalities and carry out the required actions based on the source article.

Before starting the proposed architecture and design of the TOTEM defined SDK, a detailed review of TOTEM theory is conducted. The expression on the workflow of the architecture proposed in TOTEM will be followed by an indication of how the Blockchain Technology and MapReduce concepts are integrated into the theory of TOTEM. Finally, the architecture in the developed TOTEM defined SDK will be explained.

It should be noted, of course, that the content presented in the Theory of TOTEM 3.1 and the Workflow in the TOTEM architecture 3.2 sections of this chapter, is to help potential readers figure out the background of the TOTEM SDK, and obtained from [1], where the whole idea is expressed.

3.1 Theory of TOTEM

Token for controlled computation (TOTEM) is a novel idea, introducing a new way to analyze data. TOTEM aimed at bringing the computation to data to avoid the drawbacks associated with the traditional methods for data analysis, in particular

higher bandwidth demands and security breaches. Analysis in these methods is conventionally carried out by request for moving data across the network. This leads to considerably high bandwidth consumption. In addition to that, asset owners and data providers are often not willing to expose their data visible to everyone in the network. This makes their data privacy and security vulnerable to the possibility of being cyber attacked.

TOTEM introduces a new approach, focusing on the establishment of a framework to integrate Blockchain Technology with Big Data Systems, which can utilize the advantages of both technologies effectively. As discussed in Chapter 2, the Hadoop framework allows for parallelized computation on large data sets through its underlying MapReduce library. Figure 3.1 from [30] illustrates parallel computation on a large data set of input data.

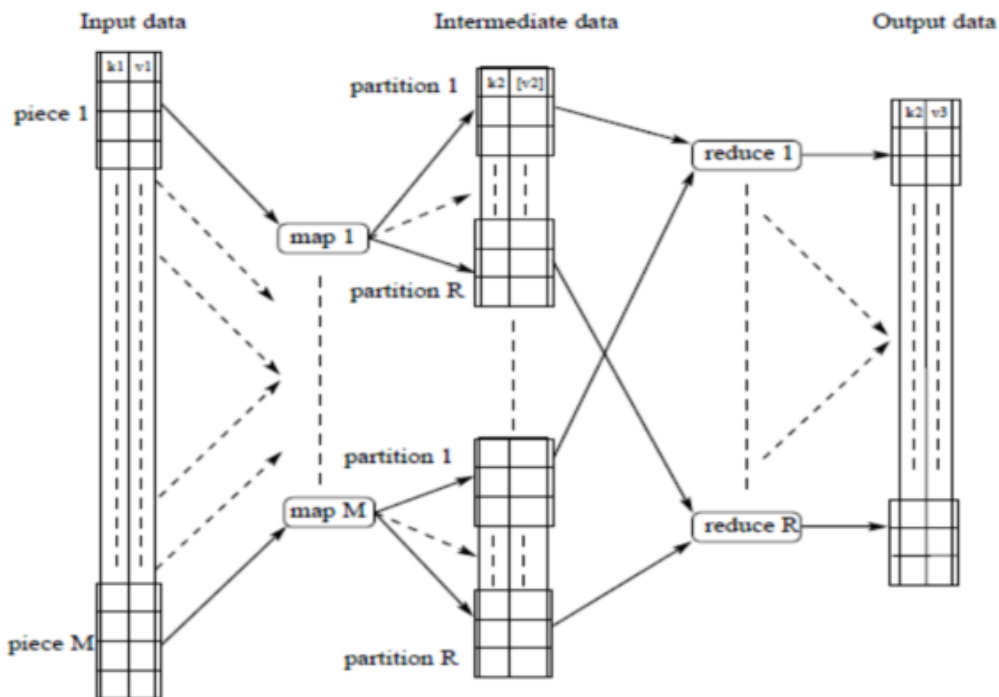


FIGURE 3.1: Parallel MapReduce computations [30]

The parallelism is performed through a parallel map over input data in the first place. Then, the intermediate data are grouped parallelly by key, as needed for the reduce phase. Parallel reduction per group is carried out ultimately to produce the output data [30].

On the other hand, Blockchain is restricted by block size and creation frequency; consequently, it is incapable of handling large data sets or parallel processing. However, Blockchain has proved to be a highly tamper-proof, secure, and reliable option for recording transactions. TOTEM combines the strength of these two technologies,

which can complement each other, to develop a new approach. The architecture illustrated in Figure 3.2 shows the integration of the Hyperledger Fabric and Hadoop framework. In general, TOTEM architecture comprised of three fundamental layers: Blockchain, Storage layer, and Computation layer.

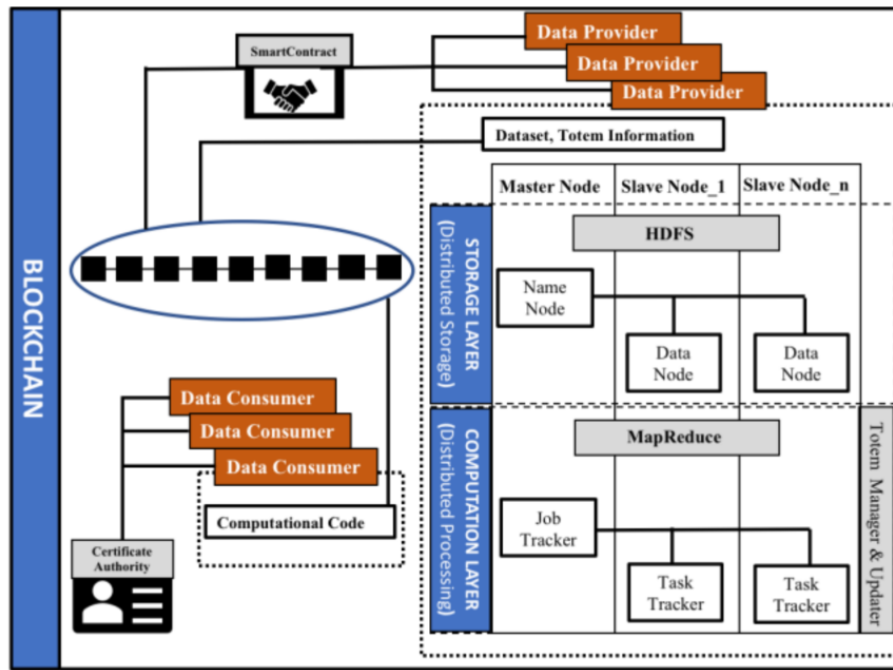


FIGURE 3.2: TOTEM architecture [1]

In the blockchain consortium, Data Providers and Data Consumers play the most critical roles and are considered as the main characters. Data Providers, as the name indicates, are the owner of the data. They provide data by linking their Big Data resources to the system and publishing the meta-data of these data sets to the Blockchain. Conversely, Data Consumers are the entities who use the provided data, after being authenticated to be given access to the data resources. Data Consumers can execute their computational code on the available data sets.

A further important actor in the blockchain layer is the Smart Contract. Smart Contracts are deployed by the Data Providers and primarily responsible for the validation and evaluation of the computational code submitted by the Data Consumers. Smart Contract ensures more transparent facilitation by restricting the computation complexity. In other words, Smart Contract monitors the malware or infinite loops in the submitted code and excludes them from execution. With various Data Providers operating within the blockchain consortium, having a standardized monitoring process is of importance. This process should represent an inclusive effort from all Data Providers serving one purpose: validation of the consumer's code to limit the computation complexity and prevent malicious functions. The new concept *TOTEM* proposed in [1] is introduced for proper validation of the computational code.

TOTEM is an entity that monitors the complexity of the Consumers' submitted code and its potential malicious bugs, to stop network latencies. Totem resembles the concept of gas in Ethereum [31], and similarly, it will be assigned to users depending on their computational needs. These authenticated consumers use their totems during the execution time so that the more complicated their computational codes are, the more totems they would need for execution. The usage of totem will be continuously controlled during computation until either the computation process finishes successfully or totem value exhausts.

The other additional layers in the TOTEM architecture are the Storage layer and the Computation layer. The Storage layer is used to store the actual data and represents the Hadoop Distributed File System (HDFS). In contrast, the Computation layer contains the MapReduce framework to do distributed processing.

Totem manager and updater are also introduced to handle totem consumption and update its status by the execution of each opcode.

3.2 The workflow in the TOTEM architecture

The architecture discussed in the Theory of TOTEM 3.1 gives a general overview of the synthesis of TOTEM structure and how they function. However, this section is written to show the entire process in detail, starting from Data Consumers' authentication process and ending with the computation of their submitted codes and further results. The workflow presented in the following is divided into two sub-sections: The Data Consumers' side and Data Providers' side.

3.2.1 Data Consumers Side

The workflow begins with the authorization process. The Data Consumers can gain access to the data sets, provided under the blockchain consortium when they are authorized, and this happens through enrolling in the consortium.

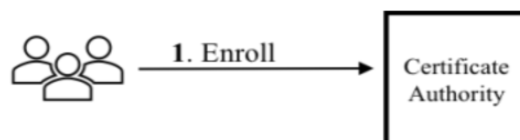


FIGURE 3.3: Enrollment in the blockchain consortium [1]

As soon as the Certificate Authority (CA) is obtained, and Data Consumers are authorized, they will be allowed to request for the meta-data and enabling the computing facility provided by the Data Providers. This request is essential for Data Consumers as they can perform their computation and estimate how many totems will

be required for it. However, there is no specific way of acquiring totems from the blockchain consortium suggested in [1] since it is highly dependent on the underlying network architecture. In the next step, the Data Consumer will get a response containing a key to access the data and the required totem for computation.

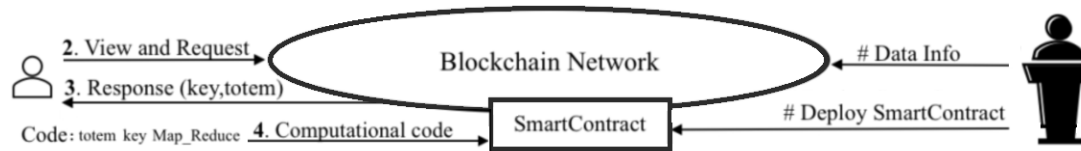


FIGURE 3.4: Data Consumer Side of the TOTEM workflow [1]

In the schematic diagram depicted in Figure 3.4, steps 2, 3, and 4 of the workflow, as well as the relations between involved entities, are shown. In step 4, the computational code will be provided by the Data Consumer for deployment and execution on the requested data. The computational code must correspond to the Map_Reduce pattern of the SDK defined for the TOTEM. Further to this, the consumer provides the key and the required totem for execution in order to get permission to perform their code. Then, the smart contract which was deployed by the Data Provider will perform an evaluation on the consumer code. It processes the submitted code and determines whether there are sufficient totems provided by the Data Consumer regarding the computational needs of the submitted code. Script 3.1 indicates a preliminary check of the code deployed by a Data Consumer (Code*) written in pseudo-code.

SCRIPT 3.1: Smart Contract preliminary check in pseudo-code [1]

```

Evaluate (Code*)
E = Estimated totem
if (available totem >= E) {
    Allow Computation
} else {
    Terminate
}

```

The Evaluate function uses a Totem Estimator Table (TET) to evaluate an approximate value of required totems. The code is allowed for computation if the available amount of totem associated with the code is more than or equal to the evaluated value. The steps followed by the approval for computation will be dealt with in more detail in the Data Providers Side 3.2.2.

Totem Estimator Table

Totem is an entity that monitors the complexity of the Consumers' submitted code. The given opcodes should be evaluated in such a way that the amount of totem required to perform each opcode is shared among all Data Providers in the blockchain consortium. Totem Estimator Table (TET) accounts for a table that describes standard

totem values to perform each of the defined opcodes. As mentioned earlier, the smart contract is executed before the computation of the consumer code to estimate the required totem and then check the Data Consumer's balance. This estimation of the submitted code is carried out by using the information within the TET. The estimator table should include all the defined opcodes on the data. Table 3.1 shows an example of a TET with different operators and data types and their corresponding weights.

The estimate of the totem is made through a function of opcodes and data types. Considering the type of data is essential, as various data types require various spaces to store the data. For instance, the addition of two values with data type double requires more bytes than two integer values. By assuming the Data Consumer code as $Code^*$, comprising several opcodes, each represented by C , we can write: $Code : \{c_1, c_2, c_3, \dots, c_n\}$. Opcodes can be defined operators, such as assignment, arithmetic, logical, bitwise, etc with the corresponding weights set $Weight : \{w_1, w_2, w_3, \dots, w_n\}$. Similarly, if a set of supported data types is assumed as $Datatype : \{d_1, d_2, d_3, \dots, d_n\}$, then the corresponding amount of bytes required to store these data types can be written as $Byte : \{b_1, b_2, b_3, \dots, b_n\}$.

Regarding the above assumptions, the general formula for the Estimated Totem (ET) required to perform $Code^*$ on the data can be expressed as:

$$ET = \sum_{i=1}^n Weight(c_i) \times Byte(d_i)$$

where n = the number of opcodes in $Code^*$, $c_i \in Code$ and $d_i \in Datatype$.

Operator		Datatype	
Operator	Weight	Datatype	Byte
Arithmetic	2	Integer	4
Assignment	1	Double	8
Logical	2	Float	4
Relational	1	Date	3
Read	1	#	#
#	#	#	#

TABLE 3.1: An example of Totem Estimator Table [1]

3.2.2 Data Providers Side

The further steps of the TOTEM architecture workflow proceed in the Data Providers' side when the consumer code is allowed for actual computation. This computation is carried out in a Customized Computational Framework (CCF), which resembles

the architecture used in Hadoop with some modifications. Figure 3.5 shows the Customised Computational Framework employed in the TOTEM architecture and outlines the relations among the comprising components. As it is shown, the CCF start performing the requested computation after the preliminary check implemented in the smart contract 3.1 passes. This stage continues following the steps described below until the corresponding results are produced. More details on the CCF architecture will also be given at the end of this subsection.

Within the new computational framework, one totem manager and one totem updater are attached to the master node and each of the slave nodes, respectively. Following each execution of the given opcodes, totem updaters receive reports from each peer slave node about the performed opcode. Once the report is received, each totem updater sends a signal to the totem manager of the master node, which is responsible for the calculation of the available balance. At this point, the totem manager computes the remained totem after collecting all reports from the updaters, and then sends an update back to the totem updaters. By getting the response from the totem manager, updaters signal their attached slave nodes to move on to the next opcode execution.

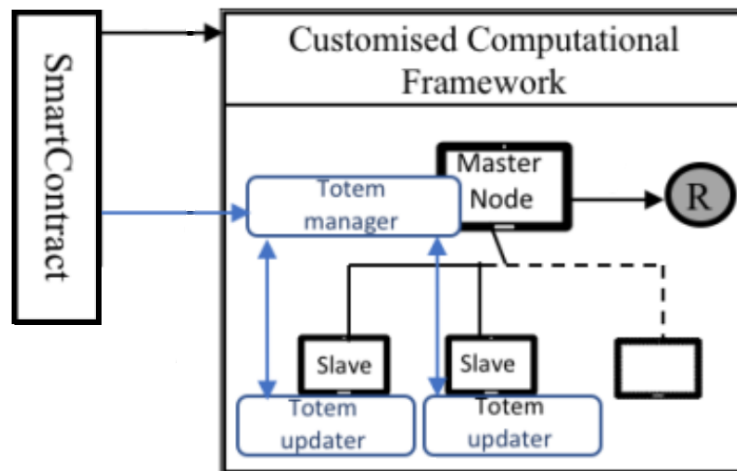


FIGURE 3.5: Data Provider Side of the TOTEM workflow [1]

This cycle will repeat either until the completion of executing a chain of map/reduce function or the available totem exhausts, and the system reaches the "Out of totem" status. In case the prior happens, updaters send the last report to the totem manager. However, if the latter takes place, the totem manager will immediately signal the master node to halt the execution.

When the execution of map/reduce function has been completed, the manager sends the final amount of totem consumed during the execution to the Blockchain. For this purpose, the manager creates a transaction, which invokes the smart contract that has already been deployed to the blockchain consortium. As it is stated in Script 3.2, the

smart contract monitors the availability of totem for further computations every time and will let the master node to execute the next function in case of sufficient totem.

SCRIPT 3.2: Totem balance checker in the Smart Contract [1]

```

For each Map/Reduce function :
  if (available totem > 0) {
    Continue
  } else {
    Terminate
  }

```

Finally, when the computation on the specified data set completes or there is not enough totem for the rest of the execution, the ultimate result illustrated as R in Figure 3.5 will be delivered to the Data Consumer.

Customized Computational Framework

On the Data Providers side, the requested data, along with a framework in which the computation is performed, are provided. As discussed earlier, the architecture used in the Customized Computational Framework (CCF) resembles the Hadoop architecture. However, the authors in [1] modified the architecture in Hadoop by introducing two new entities, totem manager and totem updater. These are extended to the master node and slave nodes, respectively, to adapt the required functionalities to the framework. Figure 3.6 indicates the workflow diagram of the CCF in detail. Following the preliminary check of the totem balance, the CCF master node takes the control and starts giving map/reduce functions as well as data to the slave nodes. This is the starting point of the CCF workflow, which is described below in various stages.

The initial stage of the CCF has been divided into two steps. In **1.1**, the actual totem balance of the Data Consumer, in addition to the estimated amount of totem required to execute the computational code is passed to the totem manager. The master node performs the next computation only if the totem manager confirms the availability of totem for further executions. At the same time, the actual computational code is sent to the master node in **1.2**. The code consists of multiple map/reduce functions, representing as M and R in the figure 3.6 and considered as the custom format of TOTEM.

In the second stage, the master node gives the first map function to slaves for execution (**2.1**) and informs the totem manager about it immediately(**2.2**). After the totem manager is notified, it forwards the information regarding the available totem to all totem updaters (**3.1**). Totem updaters will also receive the performing information of the function from their corresponding slave nodes in **3.2**.

During the stage **4** of the CCF workflow, the estimated totem by totem updater, required to perform a particular function, is compared to the actual balance. Depending

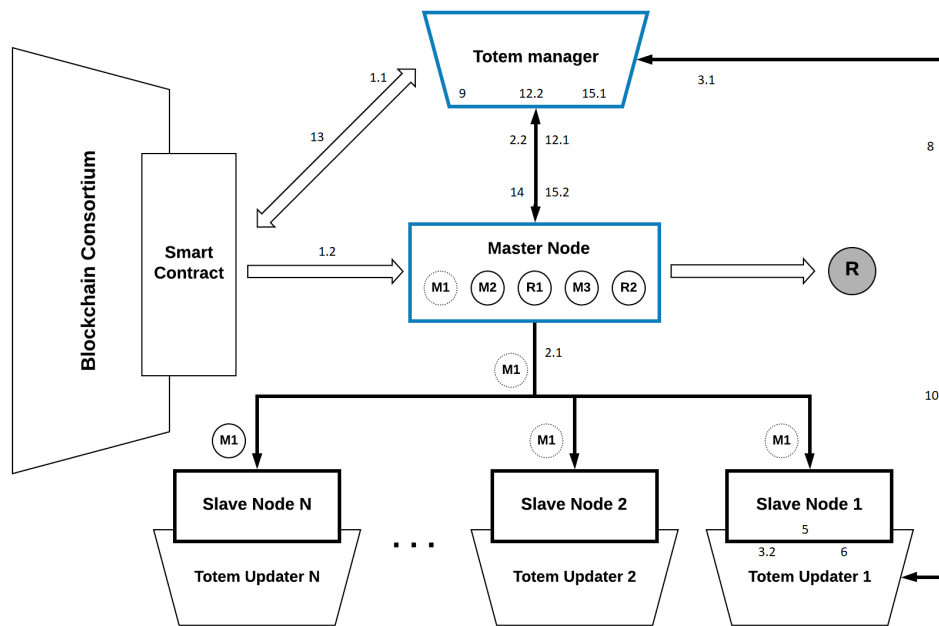


FIGURE 3.6: CCF architecture

on the sufficiency of totem for execution, it goes to either stage 5 or 15 of the workflow, which is to halt the execution immediately. Following the comparison of totem balance with the estimated value, totem updater sends an affirmative signal to the corresponding slave to begin the execution of the function (5).

As soon as each comprising opcode within the function is performed, the slave node informs the updater (6), and the updater will compute the amount of used totem for performing that opcode on the given datatype (7). Then, this amount will be updated into the totem manager to calculate the available totem (8). Script 3.3 shows the pseudo-code that is executed in the totem manager by updaters to calculate the updated totem balance and to determine whether the control should go to the next step (9):

SCRIPT 3.3: Totem balance updater in totem manager [1]

```
// totem updater in the next line :
Available totem -= Used totem
if (available totem == 0) then:
    Go to step 15.1 //"Out of totem" status
```

At this time, the calculated amount will be sent back to the updaters at the stage 10. By receiving a signal from the totem manager respecting the sufficiency of totem for further computations, updaters will send another signal to their slave nodes and confirm the continuity of executions. This will keep on again from stage 5 until the computation of the entire function completes, or the process terminates due to lack of totem (11). Soon after the completion of executing the whole function, the master node

notifies the totem manager about the successful completion of that specific function execution (12.1). Afterward, the manager updates the Blockchain with the available totem by creating a transaction within the Blockchain (12.2). In the next stage (13), the smart contract double-checks the totem balance and lets the totem manager signal the master node for further executions. However, in case of totem balance being empty, the Data Consumer will get "Out of totem" status, and the connection will be terminated.

Lastly, the master node receives a signal to release the next function. As it is shown in the figure 3.6, the next function is M2. Likewise, it goes through all steps from 2.1 to 13, the same as M1. Stage 15 is considered as the termination stage. Right after the available totem exhausts in the system, two actions will be taken place to stop the operation: "Out of totem" status announcement to the blockchain consortium (15.1), and informing the master node to terminate all executions instantly (15.2). However, if there is a final result of the last reduce function available (R2 in the figure), it will be promptly published to the blockchain consortium for the use of Data Consumer, and then the connection will be terminated.

3.3 Architecture of TOTEM defined SDK

As discussed in previous sections, the SDK plays a vital role in the architecture of TOTEM. It is aimed at enabling the consumers to submit their computational codes in the Data Providers' side. Moreover, the SDK produces an estimated value of required totem that consumer's code might cost, using the Totem Estimator Table. Ultimately, the SDK outputs a formatted code needed for the execution section. That explains why the importance of having an extensible, well-structured, and easy-to-use SDK cannot be emphasized too much.

The source code of the TOTEM SDK is released under the MIT license, and available on Github ¹. With this in mind that the defined SDK presented in this thesis is not the final version and aims at only showing the potential functionalities regarding some pre-defined rules. This section introduces the highlights and describes the layers within the architecture of the defined SDK to give a better understanding of the implementation in Chapter 4.

3.3.1 Specifications

TOTEM SDK provides a flexible toolkit for testing and deploying computational codes of Data Consumers over the Internet, setting a good starting point for researches and

¹<https://github.com/Behfar90/TOTEM-SDK>

developers to experience the new concept of Blockchain and Big Data integration. In the following, some of the highlights about TOTEM SDK is presented:

- **Community Cooperation** – the source code is put on *Github* publicly and leverage the whole community and interested developers to improve the SDK for better performance and flexibility.
- **Pure JS code** – the TOTEM SDK presented in this thesis is implemented in fully JavaScript code, including Vanilla javascript and jQuery, which is a famous JavaScript library to simplify Document Object Model (DOM). Using JS makes TOTEM SDK very efficient to deploy in web browsers regardless of operating systems and multiple hardware platforms [32].
- **Modular Design Pattern** – TOTEM SDK is designed from the beginning to conform to the Modular Design Pattern (MDP), allowing for wrapping a set of variables and functions together in a single scope. The module interface is, in fact, a piece of program that specifies which other pieces it relies on and which functionality it provides for other modules to use (its interface). By limiting the way that modules interact with each other, the whole system can be seen as a LEGO, where pieces interact through well-defined connectors called dependencies [33]. From all the benefits that MDP brings in favor of interdependent codebase, the following three are more important in the TOTEM SDK: *Maintainability*, *Namespacing* and *Reusability*. The modular pattern makes it simple to improve and update single modules and reuse them in new required places in the project. It also helps to avoid "namespacing pollution" where completely unrelated code shares global variables by providing private space for variables [34].
- **Maintainability and extensibility** – The core architecture of the presented SDK has been made to be maintainable and extensible as there are still opportunities for further enhancements. This architecture was done through the modular pattern. A well-designed module strives to reduce the dependencies on different parts of the codebase as much as possible so that it can grow and improve independently. Therefore, either adding a new module to the architecture or updating a single module would be much easier while the module is disassociated from the other pieces of code [34].

3.3.2 Layers

TOTEM defined SDK is primarily used for creating a formatted code from the user's submitted code as well as estimating its computational cost, all according to the defined rules. This analysis of the user's code and the estimation of the required totem are taken place within a three-layer architecture, which are *SDK layer*, *Controllers layer*, and *Handlers layer*. Figure 3.7 depicts these layers and describes how their comprising components are linked to each other in the architecture of the defined SDK.

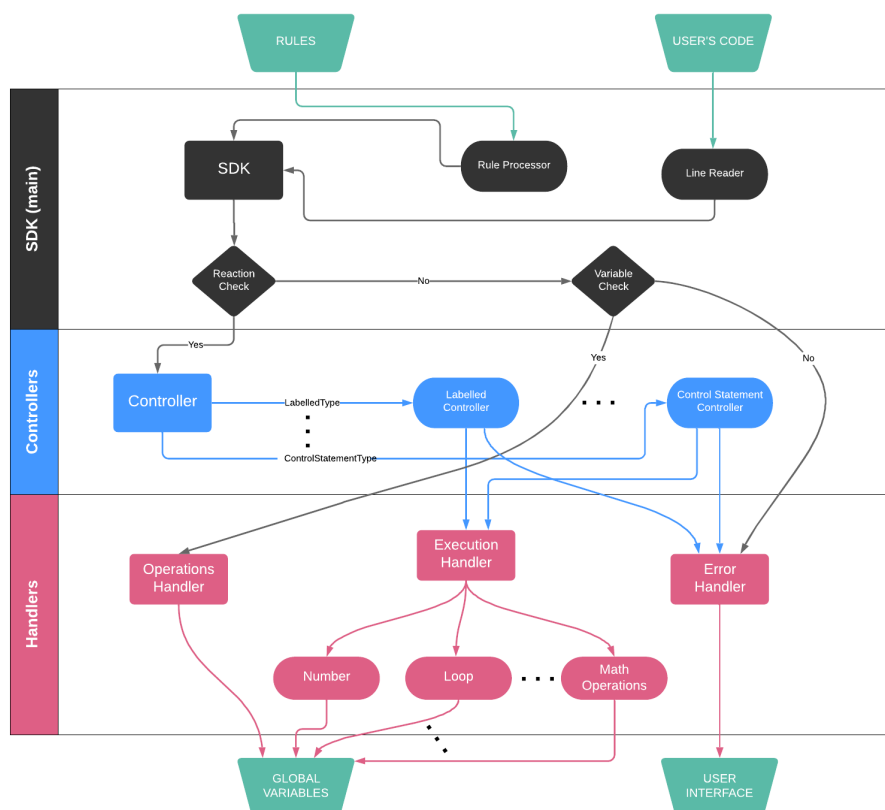


FIGURE 3.7: Workflow in the TOTEM SDK

SDK layer

The *SDK layer* accounts for the primary layer of the whole design, where all the rules and the user inputs are taken. Each line of the user input goes to the next layer if it complies with the rules. As it is illustrated in Figure 3.7, Line Reader Component (LRC) is one of the main components in the current layer in which the process of reading user input lines is carried out. Another critical component in the SDK layer is the Rule Processor Component (RPC). RPC is the only component within the whole architecture which reads, processes and extracts the proper way of handling various commands based on the defined rules, prior to any other action. Consequently, Rules is the first block of scripts that the SDK takes. This action is necessary as all the further actions occurring in the SDK layer are determined based on these rules.

Input lines go into the LRC split evenly, and the LRC divides each one into two parts of the command and the assignment part. If the command exists in the rules, the LRC will read the proper reaction to that specific command and rest will be done in the Controller layer, after conducting a check through the Reaction Checker (RC).

On the other hand, it is also possible that the input line is just another operation or mathematical equation on an already defined variable. For such conditions, there is another checker called Variable Checker (VC). VC can ignore the Controller Layer and

directs both the command as well as the assignment parts directly to the Operation Handler Component (OHC) in the Handler layer if it figures that the above condition is correct. Otherwise, Error Handler Component (EHC) will take control and throws the related error.

Controllers layer

The next layer in the TOTEM SDK architecture is the *Controllers layer*. This intermediate layer between the SDK layer and the Handler layer plays a crucial role in enabling the user's submitted code to the following global variables stored after the evaluation in the final step. On a broader level, the Controller wraps all the components, each of which handles the controlling of a specific command type defined in the rules block. Table 3.2 indicates an example of rules defined in the rules block.

COMMAND	HOW TO REACT	COMMAND TYPE	DOMAIN
Int	HandleNumber()	Labelled	Set of Z
Bool	HandleBoolean()	Labelled	Set of $\beta = \{0, 1\}$
Float	HandleNumber()	Labelled	Set of Q
For	HandleForLoop()	Control Statement	Labelled Type Vars
If	HandleIf()	Conditional Statement	Labelled Type Vars

TABLE 3.2: An example of defined rules in the TOTEM SDK

As it is indicated in Table 3.2, multiple command types might lay in the rules file. Nevertheless, for the sake of simplicity and to follow the modular pattern of the design, the Controller redirects the taken commands to their respective components. All further operations on each command type will take place in these components. Labelled Type Controller (LTC) and Control Statement Controller (CSC) are two examples of such controlling components, shown in Figure 3.7.

Following some necessary operations on the Controller inputs, such as naming convention check, splitting the assignment chunk for more evaluation, error check, etc., type controlling components consider where to point in the subsequent layer (Handlers layer). If the input is error-free, then the respective reaction function determined in the "How To React" column of the above table will be invoked. By contrast, in case of errors detected within the type controlling component, it invokes a related error in the Error Handler component to show on the User Interface (UI).

Handlers layer

The last layer in the architecture of TOTEM SDK is the *Handlers layer*, which receives requests to handle either execution, throw exceptions, or mathematical computations. The handlers layer is principally responsible for providing functions to the requesting

components within the Controllers layer. However, in a situation that a new line of command is detected as an operation on an already defined variable, that line of command within the SDK layer will be sent directly to the OHC to perform the operations in the Handlers layer.

The handlers layer consists of 3 major components in which several functions are defined. In the operations handler, all allowed operations on existing variables are managed. Taking note in the OHC, logical operations, relational assignments, and mathematical operations, such as addition, subtraction, multiplication, power, etc., are all handling functions within this component.

The error functions, on the other hand, are all wrapped up in the EHC irrespective of where the exception is being thrown. Such a component that handles all the functions, not only makes it more feasible to add, change or remove the functions in the future versions, but it also helps to avoid redundancies by calling interchangeable functions in different places of the SDK. Table 3.3 reveals various error functions defined so far in the existing version of the SDK.

FUNCTION	CALL SITE	DESCRIPTION
CommandError()	SDK	Command is not defined
HandleError()	LTC	No defined bundler to compile the command
NamingError()	LTC	Name is against the naming convention
CTRLStatementError()	CSC	An error occurred during the control statement execution
OperationError()	Execution Handler	An error occurred during the operation functions

TABLE 3.3: Error functions defined in the EHC

Execution handler accounts for another handler component with several functionalities, each of which serves to execute the output of controller components. When the processing of the input command line completes in the Controllers layer, it gets prepared for the execution stage. Depending on the call site of these executive functions, different functions could be called. Each of these executive functions handles at least one type of command defined in the rules file. In the end, the resulted outcomes from the executions will be stored in several global variables for the use of smart contracts in the blockchain consortium. Table 3.4 shows three global variables as well as their descriptions defined in the existing version of TOTEM defined SDK.

As discussed earlier, the objective of our defined SDK is not to output the final results of the submitted code as it will be done in another part of the TOTEM architecture.

TABLE 3.4: Global variables of the defined SDK

Global Variable	Description
ruleLines	A global array of map values, each representing one total row of the rules defined in rules.csv
userDefined_vars	A global object to keep track of user-defined variables and their updated values
TOTEM_operators	A global array to keep all operators and assignments within the Data Consumer's code – used to estimate the required TOTEM
mapped_executions	A global array to store the resulted mapped executions format of the Data Consumer's code - used as an input in CCF

Nevertheless, having the updated values of variables is necessary before reading the next line. The reason is that the next lines of the user's submitted code might be dependant on those variables defined earlier. As a result, considering such a global object of most updated variables and their values, which is accessible through the modular pattern in other layers, particularly in the SDK layer, is crucial.

Chapter 4

Implementation and Results

As discussed in the TOTEM architecture, the defined SDK is part of a concept purposing at enabling the data users to the requested data without exposing the whole data domain. This means everything should be handled throughout the Internet, and thus a web application is a preferable option. In other words, the application can serve as a framework for users to submit their computational codes and receive results. In this application, the authentication process, distributed processing of computation, fetching data from the distributed storage, and result creation are all done in the smart contract within the Data Providers' servers. However, the SDK can be implemented in the front-end layer while it is connected to the smart contract for totem checking.

This chapter will discuss solely the way that the existing version of TOTEM defined SDK is developed in detail. Followed by the frameworks and tools used to reach the current version, the features and functions of all constructing modules will also be explained. A User Manual sample for whatever commands that have been already defined will then be presented, and in the end, execution analysis and some testings will be performed.

4.1 Frameworks and Tools

The procedure of handling input data in the TOTEM defined SDK is based on a modular pattern. Writing the core structure of the SDK with modules helps with organization, maintenance, testing, and, most importantly, dependency management. Unlike the large traditional frameworks in which everything is done under the sun, the new approach suggests creating small modules that do one thing and one thing well [35]. For this purpose, various frameworks and tools are employed during the development to make the SDK performance as expected, which are all listed below.

1. **JavaScript JS** is one of the most effective and versatile programming languages used to extend functionality in web applications. Among all of the advantages that

JS presents, being fast for the end-user, no compilation required, being easy to test and debug, platform independency, and capability of procedural programming are the ones who make JS a superior language to write the core structure of the SDK with [36].

2. **JQuery**¹ jQuery accounts for one of the fast, small, and feature-rich libraries written by JS designed to simplify HTML DOM tree traversal and manipulation with an easy-to-use API, working across a multitude of browsers. The jQuery contains a full suite of AJAX (Asynchronous JavaScript And XML) capabilities to read/write data from/to a server without re-rendering the web page [37]. In this thesis, the jQuery AJAX is used to read the rules file into the SDK (main) layer of the defined SDK. It can also be used to write to the output for the use of further components within the TOTEM architecture.

3. **VSCode**² Visual Studio Code is a free, lightweight source-code editor made by Microsoft for use in different operational systems. VSCode has numerous useful features that support debugging, snippets, intelligent code completion, syntax highlighting, code refactoring, and embedded Git [38]. These features, in addition to the extra extensions provided with more functionalities, are the reasons why it has been used in this thesis to develop the TOTEM SDK.

4. **LiveServer**³ liveServer is a practical and functional extension for Visual Studio Code. It primarily speeds up the development process by launching a local development server with a live reload feature for static and dynamic pages. LiveServer also provides advanced support for Hypertext Transfer Protocol Secure (HTTPS), CLI for any browser (e.g., Firefox, Chrome, Nightly), and Cross-origin Resource Sharing (CORS) mechanism which allows sharing restricted resources among domains. These features might be useful in future versions of our defined SDK.

5. **Node.js** As discussed earlier in the subsection 2.3.1, Node.js is a runtime environment that compiles JS via Google's V8 JS engine to simplify the creation of multi-functional web servers using JS. Moreover, NPM is the Node Package Manager, which helps with installing various packages and resolving their dependencies effortlessly and straightforwardly. In this thesis, NPM has been used to add some functional dependencies to the current version of the defined SDK.

6. **Browserify**⁴ With the modular architectural design of the TOTEM defined SDK, there should be a module loader library to load and wire these modules in the browser because browsers still do not support that. Browserify is a tool that helps to enable developers to modularize their JavaScript codes. Browserify allows to use Node style modules statements "*require()*" and "*export*" in the browser, and brings all the NPM ecosystem off to the server and the client; simple, yet immensely powerful [39]. Thus,

¹<https://jquery.com/>

²<https://code.visualstudio.com/>

³<https://marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer>

⁴<http://browserify.org/>

developers only need to define dependencies, and then Browserify bundles it all up into a single neat and clean JS file, which can then be included in the desired project (e.g., `<script />`).

Browserify can be installed by NPM command and has been used in this thesis to implement the modular pattern among the SDK various components and functions. Figure 4.1 illustrates browsers compatibility of the syntax `export` and the `default` keyword in different devices that are used in JS modules.

	Desktop						Mobile					Node.js	
	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Android webview	Chrome for Android	Firefox for Android	Opera for Android	Safari on iOS	Samsung Internet	Node.js
<code>export</code>	61	16	60	No	47	10.1	No	61	60	44	10.3	8.0	13.2.0
<code>default</code> keyword with <code>export</code>	61	16	60	No	47	10.1	No	61	60	44	10.3	8.0	13.2.0

FIGURE 4.1: Browsers compatibility of JS modules syntaxes [40]

7. **Watchify** Watchify is another NPM module that is known as the Browserify compatible caching bundler for super-fast bundle rebuilds. It is a helpful and time-saving tool that watches source files for any changes and re-run Browserify whenever a change in the entry-point JS file or its required modules is detected [41].

8. **Exact-math** In JS, all numbers are encoded as double-precision floating-point numbers, following the international IEEE 754 standard, which stores each number in 64 bits. Figure 4.2 indicates how each number is stored as a binary fraction based on IEEE 754. However, most of the decimal fractions cannot be represented exactly following this standard, and that causes floating-point arithmetic to be not 100% precise. Exact-math library is an NPM module, consisting of various methods of math calculations, such as addition, subtraction, multiplication, division, power, rounding, etc. which gives a precise result.

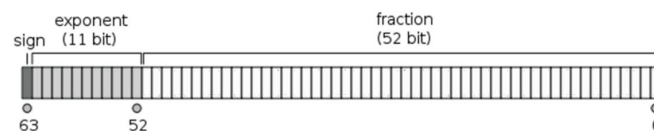


FIGURE 4.2: IEEE 754: number representation in JS

9. **Google Chrome DevTools** Google Chrome Developer Tools is a set of authoring, debugging, and developing tools built directly into the Google Chrome browser. DevTools helps significantly to edit projects on-the-fly and to diagnose problems quickly [42]. Chrome DevTools consists of several panels, including Elements panel, Console

panel, Sources panel, Network panel, Performance panel, Memory panel, Application panel, and Security panel. Using these panels, most importantly, the Console panel, not only have assisted a lot during the development of the defined SDK, but they are also employed to show the results and to perform testings on the execution of each layer.

10. **Github** Github is a code sharing and publishing service which provides hosting for software development version control using a distributed control system for tracking changes, called Git. As a command-line tool, Git gives access control and several collaboration features to developers for each project, which is stored in a *repository*. The defined SDK existing version is stored in a public repository of the author's Github account ⁵ under MIT license for further contributions.

4.2 Coding Explication

The developed SDK is implemented in a way that hypothetical Data Consumers are allowed to submit their computational codes and receive reactions. It is also worth noting that the assumption is, consumers have already gone through the authorization process which is outside the scope of this thesis. For this purpose, a simple HTML file is designed in which the SDK is imported as a bundled JS file using a `<script>` tag.

4.2.1 Environment Setup

To establish the environment as described, the first step is to initialize an NPM package. Script 4.1 reveals the initial steps, such as creating the package, adding dependencies, etc. in the root directory. Initializing a package requires to have Node and NPM already installed on the machine. As indicated in 4.1, line 1 generates a *package.json* file in the root project. This file is updated when adding further dependencies during the development process.

SCRIPT 4.1: Initial stages of establishing the SDK environment

```
(1) $ npm init
(2) $ npm install <library> --save (--save-dev)
(3) $ mkdir src
    $ cd src //To change directory into SRC
(4) $ echo > index.html
```

After the package is created, line 2 adds the library to the **dependencies** section of the *package.json* file. However, there is another section, **devDependencies**, within the *package.json* file. The difference between these two sections is that **dependencies** are modules required at run-time, while **devDependencies** contains modules which are

⁵<https://github.com/Behfar90/TOTEM-SDK>

only required during the development. Using `--save-dev` instead of `--save`, adds the library to the **devDependencies**. Followed by the installation of the desired libraries, `node_modules` directory is created automatically. The `package.json` file, in fact, defines which libraries are installed into the `node_modules` when `npm install` is run. After installation, having a `SRC` folder in which all the development occurs is of preference (line 3).

Line 4 shows creating an HTML file in the SRC directory. As discussed earlier, an HTML file can be a realistic option to show the interaction between the hypothetical user and the SDK. This includes both receiving the user's computational code and displaying feedback to the user, such as displaying errors, results, etc. In this matter, the SDK functions and checkers are imported to the HTML, and the SDK will be executed over the user's code through a triggering event (e.g., a button `onClick`). However, another option would be to store the computational code as a separate file and then execute the SDK over it. In this thesis, the prior was chosen as it offers a better way of handling feedback to the user.

SCRIPT 4.2: The body of the `index.html` in pseudo-code

```
<Body of the page>
  <Divison of the page>
    <TextArea id="userCode" />
    <Button type="button"> Run </Button>
  </Divison of the page>
  <Paragraph id="error" color="red" />
  <Script src="index.js" /> //Bundled JS file by browserify
</Body of the page>
```

Script 4.2 shows the body of the HTML file described above. In this simple file, there is a division (using a `<div>` tag) to place a text area and a button. Within the text area, users can write their computational codes according to sort of rules and then submit them for further analysis through the button. In addition to that, there is also a paragraph to show the errors associated with the users' computational codes. The errors are thrown when a statement does not comply with the defined regulations and rules. In the end, there is a script tag used to import the SDK and enable it to the button `onClick` event handler.

The `index.js` file, indicated in the Script 4.2, is the bundled JS file generated by the browserify to serve up to the browser in a single script tag. Browserify is recursively analyzing the `require()` calls in the app to build a bundle and Script 4.3 shows how simply the bundled file is created using the NPM Scripts.

SCRIPT 4.3: NPM script to build the browserify bundled JS file

```
"build": "browserify ./src/js/SDK.js -o ./src/index.js"
```

NPM scripts are part of `package.json` file and used to automate repetitive tasks.

Script 4.3 specifies that running `$npm run build` executes the defined browserify command which takes the main entry point (`SDK.js`) and bundles it to the output file (`index.js`). Likewise, a similar script can be used by watchify to monitor the changes in the `SDK.js` as well as its dependencies and automatically rebuilds the bundle. Script 4.4 shows the NPM script that keeps an eye on the files. This is quite useful during development as developers only need to reload the browser.

SCRIPT 4.4: NPM script to watch the changes in JS files

```
"watch": "watchify ./src/js/SDK.js -o ./src/index.js"
```

4.2.2 Main Functionalities

Before taking the consumer's submitted code and estimating the number of required totem for its execution, fetching the command rules and their corresponding reactions defined in the `rules.csv` file is necessary. This is done through the **jQuery AJAX call**, which accounts for one of the important functions implemented in the defined SDK. As Script 4.5 illustrates, this AJAX GET call sits in the `ready()` method of the document within the main layer where all functionalities are bundled. The `ready()` event occurs when the DOM has been loaded, and it is the best place to import the prerequisites for the rest of the functions.

SCRIPT 4.5: AJAX callback to GET the rules file in pseudo-code

```
When the document is ready(Function():
  Ajax({
    type: "GET"
    url: "address to the rules.csv"
    datatype: "text"
    success: call ProcessRules(rules) function
  }));
```

As it is shown above, the AJAX callback has four basic features. Type indicates the HTTP method to use for the request (e.g., "POST", "PUT"). URL is a string containing the Uniform Resource Locator (URL) to which the request is sent. A dataType is what is expected back from the server, and success shows the function to be called if the request succeeds.

ProcessRules is the core function inside the AJAX callback, aiming at processing each rule and storing all as an array of map values. Script 4.6 describes the `ProcessRules()` function statements in pseudo-code. As mentioned earlier, `ruleLines` is one of the global variables that keep important data for use in further computations. The `ProcessRules()` function includes a set of statements to perform its primary task which is to fill the `ruleLines` global variable. In more details, `ProcessRules()` takes all data within the `rules.csv` file, then splits the rules and header and for each rule, pushes

each header and its corresponding data to a map. Ultimately, each map value becomes one element of the ruleLines array. The reason for representing each rule as a map value is that extracting values using their paired keys is more straightforward and secure than tracking single values based on their positions within an array. Furthermore, having a map has several benefits over using an object as it is ordered and iterable and accepts any type of data as a key [43].

SCRIPT 4.6: ProcessRules function in pseudo-code

```
Function ProcessRules(rules):
  Set allTextLines to an array of splitted rules
  Set headers to an array of splitted allTextLines[0] as headers
  FOR each of the allTextLines elements
    Set data to an array of splitted each rule by comma
    IF length of headers = length of data THEN
      Set ruleMap to an empty map
      FOR each of the headers elements
        Push (header[j], data[j]) to the ruleMap
      ENDFOR
      Push ruleMap to the global var ruleLines
    ENDIF
  ENDFOR
```

lineReader is the other fundamental procedure implemented in the existing version of the TOTEM defined SDK. This function enables the consumer's submitted lines of code separately and in succession to the lower layers of the defined SDK (Controllers layer and Handlers layer) if a reaction is set out to each. In the lineReader function, each line of code is broken down into two parts, command and assignment. Then, based on the function name of the correlative reaction to each command, lineReader routes each line to either the controller or handler functions. This task is done by a local function, called getRuleValues which iterates through the ruleLines global variable to return the related information about a command.

SCRIPT 4.7: lineReader function in pseudo-code

```
Function lineReader(line of code):
  Set cmd to the command part of each line of code
  Set assignment to the rest of the line of code
  Set reaction to the output of getRuleValues(cmd,"HOWTOREACT")
  Set isVar to true if cmd is an already defined variable
  IF reaction exists and defined THEN
    Go to the Controller
  ELSE
    IF isVar = true THEN
      Go to the execution handler
    ELSE
      Go to the error handler
    ENDIF
  ENDIF
```

Script 4.7 describes how the `lineReader` function operates and routes the Data Consumers' computational codes line by line to the successive layers. Based on the SDK regulations, which will be presented in the Appendix A, if a line includes a command, then the command comes first. Unless it is arithmetic, logical or relational operation over an already defined variable in which no defined command is in the command part of the line. This justifies why the `isVar` checking is necessary when there is no known command or no reaction to the command part of a line.

SCRIPT 4.8: Listener function for button "click" event in pseudo-code

```

Button.addEventListener("click", function():
    Set userCodeValue to the value of the textArea(#userCode)
    Set codeValueLines to an array of splitted submitted code by line
    Set loopLine to an empty array
    Set statementLine to an empty string
    FOR EACH line IN codeValueLines
        IF line command is of control statement type THEN
            Push line and next lines within "{}" to the loopLine
            Delete all pushed lines from codeValueLines
            Set statementLine to concatenated lines in loopLine by ";"
            Replace statementLine with deleted lines in codeValueLines
        ENDIF
        IF statementLine is empty THEN
            Execute lineReader(line)
        ELSE
            Execute lineReader(statementLine)
        ENDIF
    ENDFOR
);

```

The `addEventListener` for the event target button accounts for the other primary functionality implemented in the TOTEM SDK. This method sets up a function that will be called whenever the specified event is delivered to the target. This procedure, which is shown thoroughly in Script 4.8 in pseudo-code, is executed before the `lineReader` function operates. The function specified above is triggered when users submit their codes by clicking the button. The button "*click*" event here, resembles the circumstance in which users desire to receive results after completion of their codes.

The function indicated in Script 4.8, takes the entire submitted code, separates it by line and stores each line as an element of a pre-defined local array (`codeValueLines`). The reason is to deliver each command separately in one piece to the `lineReader` function. However, some commands (e.g., control statements) are more than a single line of code. Therefore, the consecutive lines related to a single command should all be considered as one. That is what the first if statement manages to do. In the following, the second if statement checks the next line of command is either an initial splitted line or a block of lines belonged to a sole command. Then, it calls the `lineReader` function with the proper argument.

4.3 Execution and Testing

To verify that the developed SDK can take the submitted code, analyze it, perform the error investigation and produce the expected results, the console panel in the Google Chrome DevTools was used to capture the messages and show the outputs. It should also be noted that all the executions and testings which are presented in this section are conducted using three defined commands INT, FLOAT, and FOR loop. The first screenshot, Figure 4.3, displays the ruleLines global variable, the array of rules obtained from the rules.csv file. In the figure, one can see the formatted map elements of the array, each representing one rule that users are allowed to use. Once the page DOM is ready for JavaScript code to execute, jQuery detects this readiness and performs an asynchronous HTTP request to get the rules. Every rule contains several (key => value) tuples equal to the number of columns in the rules.csv file.

```

Rules:
(3) [Map(4), Map(4), Map(4)]
  0: Map(4) {"COMMAND" => "int", "HOWTOREACT" => "handleNumber", "CMDTYPE" => "labelled", "DOMAIN" => "set of Z"}
  1: Map(4) {"COMMAND" => "float", "HOWTOREACT" => "handleNumber", "CMDTYPE" => "labelled", "DOMAIN" => "set of Q"}
  2: Map(4)
    [[Entries]]
      0: {"COMMAND" => "for"}
      1: {"HOWTOREACT" => "handleForLoop"}
      2: {"CMDTYPE" => "controlStatement"}
      3: {"DOMAIN" => "labelledVars"}
    size: (...)
    __proto__: Map
    length: 3
    __proto__: Array(0)
  
```

FIGURE 4.3: The ruleLines global variable: array of rules

To make the tests more representative, and be able to achieve a better judgment on the SDK performance, two scenarios are defined, which include all types of commands. Script 4.9 depicts two scenarios for a user's submitted code, which within the scenario **A**, an Integer and a Float number values change within a For loop without any error. While the scenario **B** shows a code containing various types of errors in every line of it.

SCRIPT 4.9: Testing scenarios of user's submitted code: A) error-free
B) error in all lines

<pre> int a = 1, b = 2 int c = add(a, b) float d = 4.1 for (i=1; i<4; i++) { c = add(c, i) d = sub(d, 0.1) } </pre>	<pre> integer a = 1 float b% = 4.1 int c = add(3,d) int d = sub(3) for (i=1; i<4) { c = add(c, i) } </pre>
A	B

Figure 4.4 displays the results of two embedded testings in the scenario A, one in the lineReader function and the other one placed at the end of execution to show the

value of *userDefined_vars* global variable. As the figure illustrates, the SDK detects the type and the corresponding reaction to each block of command after being processed in the *lineReader* function. Moreover, the value of each defined variable is stored at *userDefined_vars* and gets updated by each command block execution. These data are the input of the Controllers layer. Similarly, in the second testing scenario, the type and the reaction to command blocks with defined commands, excluding the first line as it is undefined, are detected by the SDK. However, there will be no variables stored in the *userDefined_vars* due to the exceptions thrown by the error handler in every line. Figure 4.5 displays the errors associated with the code in scenario B.

3	Type: labelled , Reaction: handleNumber	index.js:10671
	Type: controlStatement , Reaction: handleForLoop	index.js:10671
	vars: ▶ {a: 1, b: 2, c: 9, d: 3.8, i: 4}	index.js:10711

FIGURE 4.4: Controllers layer input data as well as the final values of variables in testing scenario A

Command integer is not defined/allowed.
Variable name b% is not allowed/defined.
Variable d is not defined.
There must be 2 arguments in operating functions.
For loop must have 3 statements in definition.

FIGURE 4.5: The errors associated with the code in testing scenario B

Two more tests are conducted before the termination of the execution for the testing scenarios in 4.9. The objective of performing these tests is to figure the *TOTEM_operators* and *mapped_executions* global variables that contain required data for both totem estimation and the CCF. As mentioned earlier, *TOTEM_operators* stores all the assignments and operators within the user's code. This information is then used to estimate the amount of totem required to perform the computation using a pre-defined TET and execute the code in the computation part. Figure 4.6 shows the elements stored in this global array at the end of execution in scenario A.

```

operators: index.js:10712
(18) ["INTassign", "INTassign", "MathOperation", "FLOATassign", "INTassign",
n", "RelationalOperation", "MathOperation", "MathOperation", "MathOperatio
▶ n", "RelationalOperation", "MathOperation", "MathOperation", "MathOperatio
n", "RelationalOperation", "MathOperation", "MathOperation", "MathOperatio
n", "RelationalOperation"]

```

FIGURE 4.6: The *TOTEM_operators* global array elements after the execution in testing scenario A

Figure 4.7 also displays the *mapped_executions* global array when the execution terminates in the testing scenario A. As depicted, each element of this array represents a single execution in the submitted code, mapped together for the use of CCF at the further levels, e.g., computation framework, which is beyond the scope of this thesis.


```
mapped executions: index.js:10713  
(18) ["int a=1", "int b=2", "c=add(a,b)", "float d=4.1", "int i=1", "i<4",  
▶ "c=add(c,i)", "d=sub(d,0.1)", "i=add(i,1)", "i<4", "c=add(c,i)", "d=sub(d,  
0.1)", "i=add(i,1)", "i<4", "c=add(c,i)", "d=sub(d,0.1)", "i=add(i,1)", "i<  
4"]
```

FIGURE 4.7: The `mapped_executions` global array elements after the execution in testing scenario A

In the testing scenario B, conversely, both *TOTEM_operators* and *mapped_executions* remain empty after the execution. The reason is when an error occurs; the EHC takes control over the Controller or the Execution handler. Thus, that line of command will not be executed and ultimately stored in either of these arrays.

Chapter 5

Conclusion and Future Work

The main objective of this thesis was to implement the defined SDK within the TOTEM computational system with a particular focus on devising a TOTEM language and regulations. This has been completed, and two operational scenarios emulating Data Consumers' computational codes, have been implemented. These test-beds have been conducted only to demonstrate the stability of the SDK framework, even though there is still room for the addition of new operations and allowed data types in the rules.csv file.

Using a multi-layer architecture in designing the TOTEM defined SDK and eventually deploy all functionalities within this architecture not only helps to maintain the level of data security high, but it also makes further enhancements easy. However, there might be some concerns that the tiered structure might add unexpected complexity to the project and increase the maintenance costs, as well as the risk of failure. The author believes that the costs of higher complexity compared to the costs of new machines (for additional tiers) and network bandwidth are relatively low.

From an operational point of view, using JavaScript and Node.js to create the SDK and what it aims for, which is to make map-reduce formatted computational code specific to the TOTEM platform, was a practical choice that should be considered in further versions of the SDK. In TOTEM, the computation takes place where the data is located, and this might justify web applications as the best servers to it by giving the ability to streamline the operations. Additionally, JavaScript and Node.js are the best tools to develop the SDK as a web application as they make code deployment and executions quicker and easier and also assist in much faster data transmission between the server and the client.

For future work, it would be necessary to extend the number of data types and operations that are allowed to use by Data Consumers while submitting their computational codes. The next thing that could be experimented further is to add a framework for assigning totem to the users, and also an estimator table to emulate the process as

described in the TOTEM workflow when there is a smart contract serving as a back-end to the SDK. Future studies could also aim to replicate results in a broader scope, e.g., creating a sole compiler that can be installed separately rather than interpreting codes deployed into the browser. Last but not least, as TOTEM accounts for a novel architecture, the author considers publishing a paper based on the work that has been done during this thesis in the near future.

List of Figures

2.1	Ethereum transaction example using gas fee [12]	5
2.2	Example Ledger: fabcar [15]	6
2.3	An overview of HDFS architecture [18]	8
2.4	The SHDFS architecture [27]	11
3.1	Parallel MapReduce computations [30]	14
3.2	TOTEM architecture [1]	15
3.3	Enrollment in the blockchain consortium [1]	16
3.4	Data Consumer Side of the TOTEM workflow [1]	17
3.5	Data Provider Side of the TOTEM workflow [1]	19
3.6	CCF architecture	21
3.7	Workflow in the TOTEM SDK	24
4.1	Browsers compatibility of JS modules syntaxes [40]	31
4.2	IEEE 754: number representation in JS	31
4.3	The ruleLines global variable: array of rules	37
4.4	Controllers layer input data as well as the final values of variables in testing scenario A	38
4.5	The errors associated with the code in testing scenario B	38
4.6	The TOTEM_operators global array elements after the execution in testing scenario A	38
4.7	The mapped_executions global array elements after the execution in testing scenario A	39

List of Scripts

2.1	Map function in pseudo-code	9
2.2	Reduce function in pseudo-code	9
3.1	Smart Contract primilinary check in pseudo-code [1]	17
3.2	Totem balance checker in the Smart Contract [1]	20
3.3	Totem balance updater in totem manager [1]	21
4.1	Initial stages of establishing the SDK environment	32
4.2	The body of the index.html in pseudo-code	33
4.3	NPM script to build the browserify bundled JS file	33
4.4	NPM script to watch the changes in JS files	34
4.5	AJAX callback to GET the rules file in pseudo-code	34
4.6	ProcessRules function in pseudo-code	35
4.7	lineReader function in pseudo-code	35
4.8	Listener function for button "click" event in pseudo-code	36
4.9	Testing scenarios of user's submitted code: A) error-free B) error in all lines	37

List of Tables

2.1	A comparison between two popular big data frameworks	7
3.1	An example of Totem Estimator Table [1]	18
3.2	An example of defined rules in the TOTEM SDK	25
3.3	Error functions defined in the EHC	26
3.4	Global variables of the defined SDK	27

Bibliography

- [1] Dhanya Therese Jose, Antorweep Chakravorty, and Chunming Rong. "TOTEM: Token for controlled computation: Integrating Blockchain with Big Data". In: *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. IEEE. 2019, pp. 1–7.
- [2] Karim R Lakhani and M Iansiti. "The truth about blockchain". In: *Harvard Business Review* 95 (2017), pp. 118–127.
- [3] Andreas Meier and Henrik Stormer. "Blockchain= Distributed Ledger+ Consensus". In: *HMD Praxis der Wirtschaftsinformatik* 55.6 (2018), pp. 1139–1154.
- [4] Stuart Haber and W Scott Stornetta. "How to time-stamp a digital document". In: *Conference on the Theory and Application of Cryptography*. Springer. 1990, pp. 437–455.
- [5] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. Tech. rep. Manubot, 2019.
- [6] Gareth W Peters and Efstathios Panayi. "Understanding modern banking ledgers through blockchain technologies: Future of transaction processing and smart contracts on the internet of money". In: *Banking beyond banks and money*. Springer, 2016, pp. 239–278.
- [7] Du Mingxiao et al. "A review on consensus algorithm of blockchain". In: *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE. 2017, pp. 2567–2572.
- [8] Nick Szabo. "The idea of smart contracts". In: *Nick Szabo's Papers and Concise Tutorials* 6 (1997).
- [9] Weiqin Zou et al. "Smart contract development: Challenges and opportunities". In: *IEEE Transactions on Software Engineering* (2019).
- [10] Christopher D Clack. "Smart Contract Templates: legal semantics and code validation". In: *Journal of Digital Banking* 2.4 (2018), pp. 338–352.

- [11] Lin William Cong and Zhiguo He. "Blockchain disruption and smart contracts". In: *The Review of Financial Studies* 32.5 (2019), pp. 1754–1797.
- [12] Preethi Kasireddy. *How does Ethereum work, anyway?* <https://www.preethikasireddy.com/post/how-does-ethereum-work-anyway>. 2017.
- [13] Elli Androulaki et al. "Hyperledger fabric: a distributed operating system for permissioned blockchains". In: *Proceedings of the Thirteenth EuroSys Conference*. 2018, pp. 1–15.
- [14] Christian Cachin et al. "Architecture of the hyperledger blockchain fabric". In: *Workshop on distributed cryptocurrencies and consensus ledgers*. Vol. 310. 2016, p. 4.
- [15] *Ledger*. <https://hyperledger-fabric.readthedocs.io/en/latest/ledger/ledger.html>. 2020.
- [16] T Ramalingeswara Rao et al. "The big data system, components, tools, and technologies: a survey". In: *Knowledge and Information Systems* (2018), pp. 1–81.
- [17] Fadi H Gebara, H Peter Hofstee, and Kevin J Nowka. "Second-generation big data systems". In: *Computer* 48.1 (2015), pp. 36–41.
- [18] Wissem Inoubli et al. "An experimental survey on big data frameworks". In: *Future Generation Computer Systems* 86 (2018), pp. 546–564.
- [19] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: a flexible data processing tool". In: *Communications of the ACM* 53.1 (2010), pp. 72–77.
- [20] Xiayu Hua et al. "Enhancing throughput of the Hadoop Distributed File System for interaction-intensive tasks". In: *Journal of Parallel and Distributed Computing* 74.8 (2014), pp. 2770–2779.
- [21] El_Rancho. *MapReduce explained*. <https://medium.com/@francescomandru/mapreduce-explained-45a858c5ac1d>. 2019.
- [22] Simon Holm Jensen, Anders Møller, and Peter Thiemann. "Type analysis for JavaScript". In: *International Static Analysis Symposium*. Springer. 2009, pp. 238–255.
- [23] Gregor Richards et al. "An analysis of the dynamic behavior of JavaScript programs". In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2010, pp. 1–12.

- [24] Ravi Chugh et al. "Staged information flow for JavaScript". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2009, pp. 50–62.
- [25] *About Node.js, and why you should add Node.js to your skill set?* <http://blog.training.com/2016/09/about-nodejs-and-why-you-should-add.html>. 2016.
- [26] Elena Karafiloski and Anastas Mishev. "Blockchain solutions for big data challenges: A literature review". In: *IEEE EUROCON 2017-17th International Conference on Smart Technologies*. IEEE. 2017, pp. 763–768.
- [27] Deepa S Kumar and M Abdul Rahman. "Simplified HDFS architecture with blockchain distribution of metadata". In: *International Journal of Applied Engineering Research* 12.21 (2017), pp. 11374–11382.
- [28] Manuj Subhankar Sahoo and Pallav Kumar Baruah. "HBasechainDB—A Scalable Blockchain Framework on Hadoop Ecosystem". In: *Asian Conference on Supercomputing Frontiers*. Springer. 2018, pp. 18–29.
- [29] Uchi Ugobame Uchibeke et al. "Blockchain access control Ecosystem for Big Data security". In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE. 2018, pp. 1373–1378.
- [30] Madhavi Vaidya. "Parallel processing of cluster by map reduce". In: *International journal of distributed and parallel systems* 3.1 (2012), p. 167.
- [31] Vitalik Buterin et al. "A next-generation smart contract and decentralized application platform". In: *white paper* 3.37 (2014).
- [32] Xin Liu et al. "VIPLFaceNet: an open source deep face recognition SDK". In: *Frontiers of Computer Science* 11.2 (2017), pp. 208–218.
- [33] Marijn Haverbeke. *Eloquent javascript: A modern introduction to programming*. No Starch Press, 2014.
- [34] Preethi Kasireddy. *JavaScript Modules: A Beginner's Guide*. <https://www.freecodecamp.org/news/javascript-modules-a-beginner-s-guide-783f7d7a5fcc/>. 2016.
- [35] Mike Chen Ben Berman Juan Pablo Osorio Ospina Stephen Margheim Kent C. Dodds. *An Intro To Using npm and ES6 Modules for Front End Development*. <https://wesbos.com/javascript-modules>. 2015.

-
- [36] Justin Spencer. *Top 9 Advantages of JavaScript*. <https://www.markupbox.com/blog/advantages-of-javascript/>. 2017.
- [37] jQuery contributors. *jQuery API*. <https://api.jquery.com/>. 2020.
- [38] *Visual Studio Code - Open Source ("Code - OSS")*. <https://github.com/microsoft/vscode>. 2019.
- [39] Peleke Sengstacke. *Getting Started with Browserify*. <https://scotch.io/tutorials/getting-started-with-browserify>. 2016.
- [40] MDN contributors. *A background on modules*. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules>. 2020.
- [41] Per Jansson. *BROWSERIFY – MODULES FOR CLIENT SIDE JAVASCRIPT*. <https://thecuriousdeveloper.com/2015/06/07/browserify-modules-for-client-side-javascript/>. 2015.
- [42] Google Developers. *Chrome DevTools*. <https://developers.google.com/web/tools/chrome-devtools>. 2020.
- [43] Maya Shavin. *ES6 — Map vs Object — What and when?* <https://medium.com/front-end-weekly/es6-map-vs-object-what-and-when-b80621932373>. 2018.

Appendices

Appendix A

TOTEM SDK User Manual

The following tables provide an outline of the sort of areas and details that a TOTEM defined SDK user should consider, including all commands and data types already defined. They also include all allowed programming operations manuals to provide all the information and instructions necessary to enable the users to execute their codes safely and effectively.

The User Manual is not exhaustive and may be adjusted as necessary to suit the particular arrangements of specific users.

Section	Subject	Comment
1	General rules	
1.1	Content	<ul style="list-style-type: none"> - A computational code consists of a number of lines - <u>ENTER</u> separates lines from each other (except control statements which all considered as one line) - In each line, there should only be a space between the command and the rest of the line - When a line contains only more operations on already defined variables, the var names are considered as the command
1.2	Definitions	<ul style="list-style-type: none"> - Variable names including “(”, “)”, “*”, “-”, “%” and “\$” are not acceptable according to the naming convention. - All the unnecessary spaces are ignored and not considered - There must be two arguments in operating functions (e.g., math operations)

Section	Subject	Valid Forms	Comment
2	Data Types		
2.1	Int	<ol style="list-style-type: none"> 1. int x 2. int x, y 3. int x=10, y= x+3 	<ul style="list-style-type: none"> - Var name after space - Multiple vars separated by a comma - Value assignment with ‘=’. Assignment dependent on another value must start with the value, not the number

2.2	Float	<ol style="list-style-type: none"> 1. float x 2. float x, y 3. float x=4.1, y= x+0.1 	<ul style="list-style-type: none"> - Var name after space - Multiple vars separated by a comma - Value assignment with '='. Assignment dependent on another value must start with the value, not the number
3	Operations		
3.1	For	<ol style="list-style-type: none"> 1. for (i=x; i<y; i++) { // iterations } 	<ul style="list-style-type: none"> - All considered as one line - There must be 3 statements in for() separated by ';'. - Iterations within the loop are executed in number of loops

Appendix B

Configuration guide

Configuration files, dependencies, and all the JavaScript codes and functions used in this thesis can be found at <https://github.com/Behfar90/TOTEM-SDK.git>.

The HTML code, as well as the bundled SDK functions, set up and used to conduct testing scenarios, were deployed by Visual Studio Code 1.46.1, Node.js 10.15.0, NPM 6.14.5 and tested on Google Chrome 83.0.4103.116 (Official Build 64-bit).

Before starting the steps required to execute the source code, Node.js should be already installed on the machine. By installing Node through <https://nodejs.org/en/>, one automatically gets NPM installed on their machine.

Steps needed before running the project and performing the testing scenarios are as follows:

```
$ git clone https://github.com/Behfar90/TOTEM-SDK.git
$ cd TOTEM-SDK
$ git checkout add-license-1
$ git branch
$ git pull
# To open the project folder in the VSCode
$ code .
```

Now the project folder is up and running within the VSCode environment. However, any other IDE or code editor could also be used. The next step is to install the dependencies of this project, which can be seen in the *package.json* file. To install the dependencies, we can use the terminal embedded in the VSCode by pressing Ctrl + Shift + T:

```
$ npm install
# run 'npm audit fix' if vulnerabilities found among the dependencies
```

The above npm command will install all the libraries and dependencies needed to run the project. The next step is to build the bundled JavaScript index file that is imported in the HTML file. According to the scripts section of the *package.json* file, "build" bundles the *SDK.js* file in the *js* folder and creates the output file *index.js*. In addition to that, "watch" starts watchify to monitor the changes, which might not be necessary for here.

```
$ npm run build
$ npm run watch (only assists while developing)
// cd src
// start ('open' in macOS terminal) -a "Google Chrome" index.html
```

The last step is to open the *index.html* file in the browser. The last two commands show the way to open the *index.html* through the terminal. However, this way will probably cause error messages about Cross-Origin Requests (COR) that blocks the requests due to Cross-Site Attacks (CSA). As a result, we need a local server set up on the machine.

As mentioned before, Live Server extension is an easy option that can be installed via <https://marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer>. By having Live Server installed on the VSCode, one can open the project by pressing Ctrl + Shift + P and type Live Server: Open with Live Server, then click the src folder where the *index.html* is located.

Now, the project is ready, and testing scenarios mentioned in Script 4.9 can be replicated by submitting the codes in the text area and click on 'Run'.

To see the results, open the inspect elements of the Google Chrome (by pressing Ctrl + Shift + I or right-click on anywhere in the page and click Inspect) and choose the console panel to see the rules, type and corresponding reaction to each command, and the global variables.