Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Επαληθεύσιμη Κατανεμημένη Αποθήκευση με Χρήση Blockchain

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΛΥΖΙΑ-ΜΑΡΙΑ ΚΩΝΣΤΑ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2020

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Επαληθεύσιμη Κατανεμημένη Αποθήκευση με Χρήση Blockchain

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΛΥΖΙΑ-ΜΑΡΙΑ ΚΩΝΣΤΑ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 29η Οκτωβρίου 2020.

Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Γεώργιος Γκούμας Επίκουρος Καθηγητής, Ε.Μ.Π	Διονύσιος Πνευματικάτος Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2020

Αλύζια-Μαρία Κώνστα

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

Περίληψη

Σκοπός της διπλωματικής είναι η ανάπτυξη ενός συστήματος αποθήκευσης δεδομένων, το οποίο παρέχει υψηλή επίδοση ενώ ταυτόχρονα εξασφαλίζει την ασφάλεια των δεδομένων. Για την επίτευξη υψηλής επίδοσης αλλά και για την κλιμακωσιμότητα σε σχέση με τον όγκο των δεδομένων, συνήθως καταφεύγουμε σε κατανεμημένες προσεγγίσεις όπως το HDFS ή το GFS της Google. Παρόλα αυτά, καθώς πλέον δεν υπάρχει μια κεντρική αρχή που εμπιστευόμαστε αλλά σε ένα κατανεμημένο σύστημα οι κόμβοι μπορεί να τρέχουν από διαφορετικές φυσικές οντότητες, εγείρεται θέμα εμπιστοσύνης και ασφάλειας των δεδομένων. Για να αντιμετωπίσουμε το πρόβλημα αυτό, στην εργασία αυτή χρησιμοποιούμε την τεχνολογία του blockchain. Η δυνατότητα εκτέλεσης κώδικα με επαληθεύσιμο τρόπο (έξυπνα συμβόλαια) έχει οδηγήσει στην υιοθέτηση των blockchains σε μια πληθώρα εφαρμογών πέρα των κρυπτονομισμάτων. Το σύστημα που αναπτύξαμε, ενσωματώνει το Ethereum blockchain στο HDFS και εξασφαλίζει ότι κανένας κόμβος δεν μπορεί να τροποποιήσει/διαγράψει αρχεία χωρίς η ενέργεια αυτή να περάσει από το πρωτόκολλο του HDFS. Η πειραματική αξιολόγηση του συστήματος έδειξε ότι η ασφάλεια δεν έρχεται δωρεάν αλλά πρέπει να θυσιάσουμε μέρος της επίδοσης. Πιο συγκεκριμένα, πληρώνουμε σε χρόνο ένα επιπλέον χ% κατά την εισαγωγή αρχείων και ενα ψ% στη διαθεσιμότητα του συστήματος.

Λέξεις κλειδιά

Ασφαλές Σύστημα Αρχείων, Merkle Trees, Blockchain, Κατανεμμημένο Σύστημα Αρχείων, HDFS.

Abstract

The purpose of the diploma thesis is to develop a data storage system, which provides high performance while ensuring data security. To achieve high performance but also for scalability in relation to the volume of data, we usually resort to distributed approaches such as HDFS or Google GFS. However, as there is no longer a central authority that we trust but in a distributed system nodes can run from different physical entities, the issue of data trust and security arises. To deal with this problem, in this work we use blockchain technology. The ability to execute code in a verifiable way (smart contracts) has led to the adoption of blockchains in a variety of applications beyond cryptocurrencies. The system we developed integrates the Ethereum blockchain into HDFS and ensures that no node can modify / delete files without doing so through the HDFS protocol. The experimental evaluation of the system showed that security does not come for free but we have to sacrifice part of the performance. More specifically, we pay on time an additional x% when importing files and a y% on system availability.

Key words

Secure File System, Merkle Trees, Blockchain, Distributed File System, HDFS.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή Νεκτάριο Κοζύρη, για την εμπιστοσύνη που μου έδειξε για την ανάθεση της συγκεκριμένης διπλωματικής εργασίας. Θα ήθελα επίσης να ευχαριστήσω την Κατερίνα Δόκα και τον Ιωάννη Μυτιλήνη για την εξαιρετική συνεργασία, τις ιδέες, την υπομονή και την υποστήριξή τους στην εκπόνηση της εργασίας.

Τέλος, θα ήθελα να ευχαριστήσω την οικογένεια και τους φίλους μου για την υποστήριξή καθ' όλην την διάρκεια των σπουδών μου στο Εθνικό Μετσόβιο Πολυτεχνείο.

Αλύζια-Μαρία Κώνστα,

Αθήνα, 29η Οκτωβρίου 2020

Περιεχόμενα

Κατάλογος σχημάτων

Κεφάλαιο 1

Εισαγωγή

Στις μέρες μας η ανάγκη για αποθήκευση και επεξεργασία μεγάλου όγκου δεδομένων είναι απαραίτητη. Οι εφαρμογές που χρησιμοποιούν οι άνθρωποι και οι εταιρείες προϋποθέτουν την επεξεργασία και την κατοχή πολλών πληροφοριών. Οι πληροφορίες αυτές είναι αδύνατο να αποθηκευτούν και να επεξεργαστούν από μόνο έναν υπολογιστή. Η ανάγκη αυτή έφερε στο προσκήνιο τα κατανεμημένα συστήματα, πιο συγκεκριμένα τα κατανεμημένα συστήματα αρχείων για την αποθήκευση ενός τόσο μεγάλου όγκου δεδομένων. Τα κατανεμημένα συστήματα αρχείων συμπεριφέρονται ουσιαστικά σαν ένα τοπικό σύστημα αρχείων χωρίς να γίνεται αντιληπτή από τον χρήστη η κατανεμημένη φύση του συστήματος. Καθώς η λειτουργία αυτών των συστημάτων αρχείων έχει διαδοθεί πολλές εφαρμογές δημιουργούνται οι οποίες τα χρησιμοποιούν, με αποτέλεσμα απομακρυσμένοι υπολογιστές να κατέχουν κομμάτια των αρχείων μας.

Μέχρι στιγμής, οι περισσότερες εφαρμογές στο διαδίκτυο θεωρούν τους χρήστες αξιόπιστους και ρυθμίζονται από έναν κόμβο ο οποίος θεωρείται η "κεντρική εξουσία" του συστήματος. Ο κόμβος αυτός, ρυθμίζει και συντονίζει το σύστημα, επίσης θεωρείται πάντα αξιόπιστος. Η συγκέντρωση όλων των ευθυνών σε έναν κόμβο μπορεί να αποβεί μοιραία για το σύστημα, καθώς ο κόμβος αυτός μπορεί να είναι κακόβουλος. Οπότε στην περίπτωση του κατανεμημένου συστήματος αρχείων τα αρχεία μας μπορεί να διαγραφούν ή να παραποιηθούν, χωρίς να γίνει αντιληπτό μέχρι την ανάκτηση του αρχείου. Η ασφάλεια τέτοιων συστημάτων είναι πολλή σημαντική για τους χρήστες, οι οποίοι πρέπει να γνωρίζουν πως τα δεδομένα τους είναι πάντα ακέραια και έτοιμα να ανακτηθούν.

Ένα πολύ διαδεδομένο κατανεμημένο σύστημα αρχείων είναι το HDFS, το οποίο λειτουργεί με master-slave λογική. Δηλαδή, υπάρχει ένας κόμβος κεντρική εξουσία, ο Namenode, ο οποίος συντονίζει τους υπόλοιπους κόμβους-εργάτες, τους Datanodes και ελέγχει αν το σύστημα λειτουργεί σωστά. Το HDFS χρησιμοποιείται σε μια πληθώρα εφαρμογών, καθώς το λογισμικό του μπορεί να τρέξει και σε οικιακούς υπολογιστές. Όμως η απόλυτη εμπιστοσύνη που δίνεται στον Namenode δεν μπορεί πάντα να εξασφαλίσει την ορθή λειτουργία του συστήματος, καθώς θα μπορούσε να "συνεργάζεται" με κάποιον Datanode και να παρέχει λανθάνουσες πληροφορίες για την ορθή λειτουργία του συστήματος. Να αποκρύπτει δηλαδή τις τροποποιήσεις που δέχονται κάποια αρχεία. Οπότε καλούμαστε να ενισχύσουμε το σύστημα με μηχανισμούς επαλήθευσης και να δημιουργήσουμε ένα σύστημα το οποίο να μπορεί να αντιληφθεί οποιαδήποτε προσπάθεια αλλοίωσης των πληροφοριών είτε απο τον Namenode είτε από τους Datanodes.

Για να πετύχουμε την ασφάλεια των δεδομένων χρησιμοποιούμε Merkle Trees, τα οποία χρησιμοποιούνται ευρέως για την επικύρωση μεγάλων δομών δεδομένων και εκμεταλλευόμαστε τα πλεονεκτήματα του Blockchain. Η διαφάνεια των συναλλαγών και η έλλειψη κεντρικής εξουσίας που προσφέρει η τεχνολογία Blockchain, έχει διαδόσει την χρήση του σε πληθώρα εφαρμογών. Επίσης τα δεδομένα που αποθηκεύονται στο Blockchain δεν μπορούν να διαγραφούν ή να τροποποιηθούν, οπότε αποτελεί μια αξιόπιστη πηγή πληροφοριών που κανένας κόμβος του συστήματος δεν μπορεί να καταχραστεί.

1.1 Χρησιμότητα του Συστήματος

Η τεχνολογία του cloud computing κυριαρχεί στον τομέα της πληροφορικής παγκοσμίως, επιτρέποντας σε οργανισμούς και άτομα να χρησιμοποιούν απομακρυσμένους πόρους, είτε είναι αποθήκευση, υπολογισμούς ή εφαρμογές,αντί για τοπική χρήση που μπορεί να αποτελέσει δαπανηρή σε χρόνο και πόρους του συστήματος.

Οι υπηρεσίες Cloud βασίζονται κυρίως σε λίγους μεγάλους παρόχους που ενεργούν ως αξιόπιστοι κόμβοι για τη μεταφορά, αποθήκευση και επεξεργασία δεδομένων. Με αυτό τον τρόπο οι χρήστες πρέπει να δείχνουν πλήρη εμπιστοσύνη στους παρόχους για την αποθήκευση και επεξεργασία των δεδομένων τους, οι οποίοι συγκεντρώνουν μεγάλο όγκο πληροφοριών που μπορούν να χρησιμοποιήσουν προς όφελος τους. Επίσης, καταναλώνονται μεγάλα χρηματικά ποσά για την υλοποίηση τόσο μεγάλων συστημάτων, ώστε να είναι σε θέση να εξυπηρετήσουν μεγάλο αριθμό πελατών.

Ορισμένα χαρακτηριστικά που λείπουν από αυτά τα συστήματα μπορεί να προσφέρει η τεχνολογία Blockchain. Πιο συγκεκριμένα, οι ισχυρές εγγυήσεις για ακεραιότητα των δεδομένων που βασίζεται σε αποδείξεις και όχι σε απλή εμπιστοσύνη μεταξύ πελάτη και η λειτουργία ενός πλήρους decentralized συστήματος [7].

Το HDFS διαμοιράζει αρχεία σε απομακρυσμένους υπολογιστές. Συμμετέχει ένας αριθμός κόμβων και δουλευει με master-slave λογική, όπως αναφέρθηκε παραπάνω. Οι συμμετέχοντες κόμβοι μπορούν να είναι οποιοιδήποτε υπολογιστές, τους οποίους δεν έχουμε κανένα λόγο να θεωρήσουμε αξιόπιστους. Ας υποθέσουμε ότι υπάρχει ένα τέτοιο σύστημα στο οποίο θα θέλαμε να συμμετέχουν για να αποθηκεύσουμε κάποια αρχεία που είναι αδύνατο λόγω μεγέθους να αποθηκευτούν τοπικά στον υπολογιστή μας. Στο σύστημα αυτό συμμετέχουν τοπικοί υπολογιστές, απλοί χρήστες και όχι μεγάλες εταιρείες που συγκεντρώνουν όλο τον όγκο δεδομένων σε συγκεκριμένους κόμβους. Οπότε παίρνουμε την πρωτοβουλία και ανεβάζουμε τα αρχεία μας στο συγκεκριμένο HDFS Cloud. Το αρχείο χωρίζεται σε Block, τα οποία αποθηκεύονται σε απομακρυσμένους υπολογιστές. Το αρχείο μας τώρα βρίσκεται υπό την διαχείριση των συγκεκριμένων κόμβων. Οποιαδήποτε προσπάθεια για τροποποίηση ή αλλαγή των δεδομένων μας από τους Datanodes (κόμβοι που αποθηκεύουν τα αρχεία), θα μπορούσε να μην γίνει αντιληπτή. Επομένως, όταν αποφασίσουμε να ανακτήσουμε τα αρχεία μας να είναι τροποποιημένα ή ακόμη και κομμάτια του αρχείου να έχουν διαγραφεί.

Η προσθήκη του Blockchain στο HDFS προσφέρει μια λύση στο πρόβλημα της αξιοπιστίας και έναν τρόπο επικύρωσης της ακεραιότητας των δεδομένων. Τα smart contracts (προγράμματα που εκτελούνται on-chain), προσφέρουν ασφάλεια λειτουργίας και ορθότητας των αποτελεσμάτων, καθώς αποθηκεύουν το αποτελέσματα στο Blockchain και ο καθένας μπορεί να ανακτήσει και να ελέγξει το τελικό αποτέλεσμα.

1.2 Σύνοψη του συστήματος

Ο Namenode δέχεται κάθε 6 ώρες ένα BlockReport από κάθε Datanode, το report περιέχει μέχρι στιγμής μια πληθώρα πληροφοριών για την κατάσταση του Datanode. Ενισχύσαμε το report να περιέχει κάποιες ακόμη πληροφορίες ώστε να επικυρώνεται η ακεραιότητα των δεδομένων. Η κύρια ιδέα είναι ότι κάθε φορά που ενεργοποιείται ένα BlockReport, ο Datanode θα ενημερώνεται για την ανάδειξη ορισμένων αποδείξεων Merkle. Οι αποδείξεις Merkle βασίζονται στην κρυπτογράφηση των δεδομένων και χρησιμοποιούνται σε μεγάλες δομές ώστε να επικυρώνεται η ακεραιότητα τους, χωρίς να είναι απαραίτητη η κατοχή της δομής. Για κάθε μπλοκ που ο Datanode έχει αποθηκευμένο πρέπει να στείλει έναν συγκεκριμένο αριθμό αποδείξεων, το οποίο ζητείται από το Blockchain. Το Blockchain καθορίζει το πόσες και ποιές αποδείξεις θα δοθούν. Στη συνέχεια, το BlockReport αποστέλλεται στο Namenode, ο οποίος πρέπει να επικυρώσει τα αποτελέσματα. Καθώς ο Namenode λαμβάνει το BlockReport, συνδέεται με το Blockchain και ανακτά ποιές αποδείξεις υπέδειξε πως πρέπει να δοθούν από τον Datanode. Επειδή δεν είναι αξιόπιστος ο Namenode, εφαρμόσαμε μια επικύρωση On-chain και μια τοπική επικύρωση. Η on-chain επικύρωση είναι πιο ασφαλής, καθώς ενεργοποιεί την εκτέλεση ενός smart contract, αντίθετα με αυτό που παρέχεται από την τοπική επικύρωση

στο Namenode. Ο Namenode μπορεί να καταστρέψει τα αποτελέσματα και να παρέχει μη έγκυρες πληροφορίες στο σύστημα. Από την άλλη πλευρά, η εκτέλεση του smart contract αποθηκεύεται στο Blockchain και ο καθένας μπορεί να το διαβάσει και να επιβεβαιώσει την ενέργεια του Namenode ανά πάσα στιγμή. Ο αλγόριθμος είναι ο ίδιος για τις δύο περιπτώσεις. Άμα κάποια απόδειξη κριθεί λανθασμένη ο εκάστοτε Datanode υποχρεούται να διαγράψει το διεφθαρμένο Block. Σαφώς η on-chain επικύρωση προσφέρει μεγαλύτερη ασφάλεια και αξιοπιστία, αλλά έχει μεγαλύτερο κόστος στο χρόνο.

Κεφάλαιο 2

Κατανεμημένα Συστήματα

Διάφοροι ορισμοί για το τι είναι ένα κατανεμημένο σύστημα έχουν δοθεί στη βιβλιογραφία. Για τους σκοπούς μας αρκεί ο παρακάτω χαρακτηρισμός: Ένα κατανεμημένο σύστημα είναι μια συλλογή αυτόνομων υπολογιστών που εμφανίζεται στους χρήστες ως ένα ενιαίο σύστημα. Αυτός ο ορισμός αναφέρεται σε δύο χαρακτηριστικά των κατανεμημένων συστημάτων. Το πρώτο είναι ότι ένα κατανεμημένο σύστημα είναι μια συλλογή υπολογιστιστών που ο καθένας μπορεί να συμπεριφέρεται ανεξάρτητα. Ένας υπολογιστής, γενικά αναφέρεται ως κόμβος. Ένα δεύτερο στοιχείο είναι ότι οι χρήστες (είτε πρόκειται για άτομα είτε για εφαρμογές) πιστεύουν ότι ασχολούνται με ένα μόνο σύστημα. Αυτό σημαίνει ότι με τον έναν ή τον άλλο τρόπο οι αυτόνομοι κόμβοι πρέπει να συνεργαστούν. Ο τρόπος δημιουργίας αυτής της συνεργασίας βρίσκεται στο επίκεντρο της ανάπτυξης κατανεμημένων συστημάτων.

Τα κατανεμημένα συστήματα στην εποχή μας είναι πολύ διαδεδομένα και ασφαλώς χρήσιμα. Μερικές φορές είναι αναγκαία η κατασκευή ενός κατανεμημένου συστήματος, λόγω της φύσης της εφαρμογής. Σκοπός ενός κατανεμημένου συστήματος είναι η λειτουργία του ως ενιαίο σύστημα ώστε να μην γίνεται αντιληπτή από τους χρήστες η κατανεμημένη φύση του. Τα κατανεμημένα συστήματα μπορεί αν είναι υπολογιστικά, δηλαδή ένα ενιαίο πρόβλημα να διαιρείται και να εκτελείται από μια συλλογή υπολογιστών, μπορεί να είναι κατανεμημένο σύστημα πληροφοριών, για παράδειγμα μια βάση δεδομένων ή ένα κατανεμημένο σύστημα αρχείων [21].

2.1 HDFS

Το Hadoop Distributed File System (HDFS) είναι ένα σύστημα κατανεμημένων αρχείων που έχει σχεδιαστεί για να λειτουργεί και σε απλούς υπολογιστές. Έχει πολλές ομοιότητες με τα υπάρχοντα κατανεμημένα συστήματα αρχείων. Ωστόσο, οι διαφορές από άλλα κατανεμημένα συστήματα αρχείων είναι σημαντικές. Το HDFS είναι εξαιρετικά ανεκτικό σε σφάλματα και έχει σχεδιαστεί για ανάπτυξη σε υλικό χαμηλού κόστους. Το HDFS παρέχει γρήγορη πρόσβαση σε δεδομένα εφαρμογών και είναι κατάλληλο για εφαρμογές που διαθέτουν μεγάλα σύνολα δεδομένων.

Το HDFS έχει αρχιτεκτονική master / slave. Το HDFS αποτελείται από ένα μόνο NameNode, έναν κύριο διακομιστή που διαχειρίζεται το χώρο ονομάτων του συστήματος αρχείων και ρυθμίζει την πρόσβαση σε αρχεία από τους πελάτες. Επιπλέον, υπάρχει ένας αριθμός DataNodes, συνήθως ένας σε κάθε κόμβο που συμμετέχει στο σύστημα, οι οποίοι διαχειρίζονται την αποθήκευση των αρχείων του συστήματος. Το HDFS επιτρέπει την αποθήκευση δεδομένων χρήστη σε αρχεία. Εσωτερικά, ένα αρχείο χωρίζεται σε ένα ή περισσότερα μπλοκ και αυτά τα μπλοκ αποθηκεύονται σε ένα σύνολο DataNodes. Ο NameNode εκτελεί τις λειτουργίες του χώρου ονομάτων του συστήματος των αρχείων όπως άνοιγμα, κλείσιμο και μετονομασία αρχείων και καταλόγων. Καθορίζει επίσης τη χαρτογράφηση μπλοκ σε DataNodes. Οι DataNodes είναι υπεύθυνοι για την εξυπηρέτηση αιτημάτων ανάγνωσης και εγγραφής από τους πελάτες του συστήματος αρχείων. Οι DataNodes εκτελούν επίσης δημιουργία μπλοκ, διαγραφή και αντιγραφή κατόπιν εντολής του NameNode.

Ο NameNode και ο DataNode είναι λογισμικού που έχει σχεδιαστεί για να λειτουργεί σε απλούς υπολογιστές του εμπορίου. Αυτά τα μηχανήματα λειτουργούν συνήθως με λειτουργικό σύστημα GNU / Linux (OS). Το HDFS είναι γραμμένο στην γλώσσα προγραμματισμού Java, οποιοδήποτε μηχάνημα

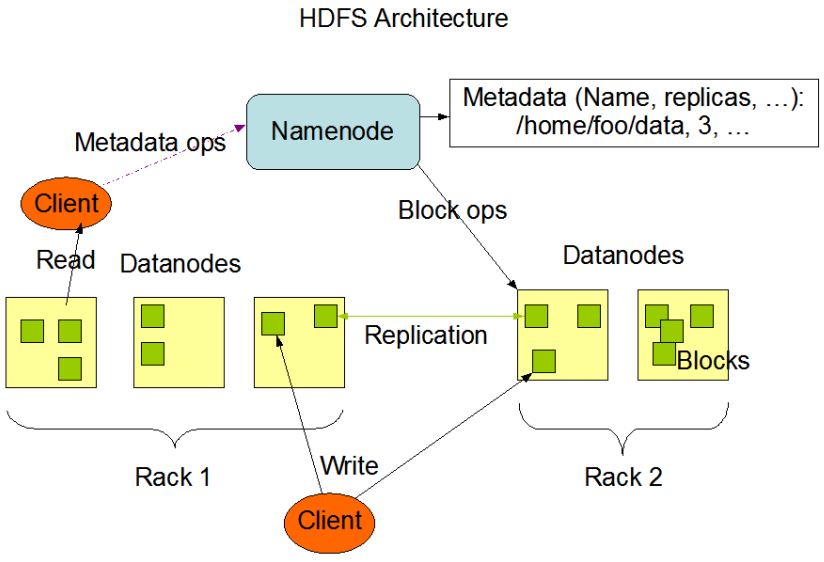
που υποστηρίζει Java μπορεί να τρέξει το HDFS. Συνήθως, το HDFS έχει μόνο έναν NameNode να τρέχει σε κάποιο μηχάνημα. Κάθε ένα από τα άλλα μηχανήματα του συστήματος εκτελεί τον κώδικα που αναπαριστά έναν DataNode. Η αρχιτεκτονική δεν αποκλείει την εκτέλεση πολλαπλών DataNodes στον ίδιο υπολογιστή. Η ύπαρξη ενός μόνο NameNode απλοποιεί σημαντικά την αρχιτεκτονική του συστήματος.

Το HDFS υποστηρίζει μια παραδοσιακή ιεραρχική οργάνωση αρχείων. Ένας χρήστης ή μια εφαρμογή μπορεί να δημιουργήσει καταλόγους και να αποθηκεύσει αρχεία μέσα σε αυτούς τους καταλόγους. Η ιεραρχία του συστήματος αρχείων είναι παρόμοια με τα περισσότερα ήδη υπάρχοντα συστήματα αρχείων. Μπορεί κανείς να δημιουργήσει και να αφαιρέσει αρχεία, να μετακινήσει ένα αρχείο από έναν κατάλογο σε έναν άλλο ή να μετονομάσει ένα αρχείο. Το HDFS δεν υποστηρίζει hardlinks ή softlinks.

Ο NameNode διατηρεί το χώρο ονομάτων του συστήματος αρχείων. Οποιαδήποτε αλλαγή στο χώρο ονομάτων του συστήματος αρχείων ή στις ιδιότητές του καταγράφεται από το NameNode. Μια εφαρμογή μπορεί να καθορίσει τον αριθμό των αντιγράφων ενός αρχείου που πρέπει να διατηρείται από το HDFS. Ο αριθμός αντιγράφων ενός αρχείου ονομάζεται συντελεστής αντιγραφής αυτού του αρχείου. Αυτές οι πληροφορίες αποθηκεύονται στο NameNode.

Το HDFS έχει σχεδιαστεί για να αποθηκεύει αξιόπιστα πολύ μεγάλα αρχεία. Αποθηκεύει κάθε αρχείο ως ακολουθία μπλοκ. Όλα τα μπλοκ σε ένα αρχείο εκτός από το τελευταίο μπλοκ έχουν το ίδιο μέγεθος. Τα μπλοκ ενός αρχείου αντιγράφονται σε πολλούς DataNodes στο σύστημα, για ανοχή σφαλμάτων. Το μέγεθος των μπλοκ και ο συντελεστής αντογραφής μπορούν να ρυθμιστούν ανά αρχείο. Μια εφαρμογή μπορεί να καθορίσει τον αριθμό των αντιγράφων ενός αρχείου. Ο συντελεστής αντιγραφής μπορεί να καθοριστεί κατά το χρόνο δημιουργίας αρχείων και μπορεί να αλλάξει αργότερα. Τα αρχεία στο HDFS γράφονται μία φορά και έχουν αυστηρά έναν συγγραφέα ανά πάσα στιγμή.

Ο NameNode λαμβάνει όλες τις αποφάσεις σχετικά με την αντιγραφή των μπλοκ. Λαμβάνει περιοδικά ένα Heartbeat και ένα Blockreport από κάθε DataNodes του συστήματος. Η λήψη ενός Heartbeat σημαίνει ότι το DataNode λειτουργεί σωστά. Ένα Blockreport περιέχει μια λίστα με όλα τα μπλοκ που είναι αποθηκευμένα σε ένα DataNode [12].



Σχήμα 2.1: Η Αρχιτεκτονική του HDFS

Κεφάλαιο 3

Merkle Trees

Ένα Merkle Tree είναι ένα δέντρο στο οποίο κάθε κόμβος φύλλο αναπαριστά το hash ενός μπλοκ δεδομένων και κάθε κόμβος που δεν είναι φύλλο αναπαριστά το hash του συνδυασμού των hashes των θυγατρικών κόμβων του. Τα Hashtrees επιτρέπουν την αποτελεσματική και ασφαλή επαλήθευση των δεδομένων μεγάλων δομών.

Ο Merkle πρότεινε ένα σχέδιο κατασκευής Merkle Tree με βάση ένα δυαδικό δέντρο στο [16]. Ένα Merkle Tree είναι ένα πλήρες δυαδικό δέντρο εξοπλισμένο με μια λειτουργία κατακερματισμού και μια ανάθεση, Φ, η οποία αντιστοιχίζει το σύνολο των κόμβων σε ένα σύνολο αντικειμένων μήκους k: n $\rightarrow$ Φ (n) $\in \{0,1\}^k$. Για τους δύο θυγατρικούς κόμβους, nleft και nright, ενός εσωτερικός κόμβου, πρέπει να ικανοποιείται η εξίσωση:

$$\Phi(\text{nparent}) = \text{hash}(\Phi(\text{nlef t}) \,||\, \Phi(\text{nright}))$$

Για κάθε φύλλο l, η τιμή Φ(l) μπορεί να επιλεγεί αυθαίρετα και, στη συνέχεια, η παραπάνω εξίσωση καθορίζει τις τιμές όλων των εσωτερικών κόμβων [22].

Διαδρομές ελέγχου ταυτότητας: Κάθε φύλλο έχει ύψος 0 και το hash δύο φύλλων έχει ύψος 1 κ.λπ. Με αυτόν τον τρόπο η ρίζα έχει ύψος H εάν το δέντρο έχει 2^H = N φύλλα. Στη συνέχεια, για κάθε φύλλο η διαδρομή ελέγχου ταυτότητας είναι το σύνολο {Auth $_h$ | 0 <=h<H}. Έτσι ένα φύλλο μπορεί να ελεγχθεί ως προς την εγκυρότητα του ως εξής: Πρώτα εφαρμόζουμε την επιλεγμένη κρυπτογραφική συνάρτηση f στο φύλλο και στο αδερφάκι του Auth_0, τότε εφαρμόζουμε την f στο αποτέλεσμα και Auth_1, και ούτω καθεξής μέχρι τη ρίζα. Εάν η υπολογιζόμενη τιμή ρίζας είναι η ίδια με την αρχική, τότε η τιμή των φύλλων γίνεται αποδεκτή ως αυθεντική. Αυτή η λειτουργία απαιτεί log $_2$(N) εφαρμογές της συνάρτησης κατακερματισμού f [5].

Merkle Proofs: Τα Merkle Proofs χρησιμοποιούνται για να εξακριβώσουμε τις παρακάτω πληροφορίες:

- Αν τα δεδομένα ανήκουν στο Merkle Tree
- Να αποδεικνύουμε την εγκυρότητα των δεδομένων ενός dataset, χωρίς να χρειάζεται να αποθηκεύουμε ολόκληρο το dataset
- η διασφάλιση της εγκυρότητας ενός συγκεκριμένου συνόλου δεδομένων που συμπεριλαμβάνεται σε ένα μεγαλύτερο σύνολο δεδομένων χωρίς να αποκαλυφθεί είτε το πλήρες σύνολο δεδομένων είτε το υποσύνολο του.

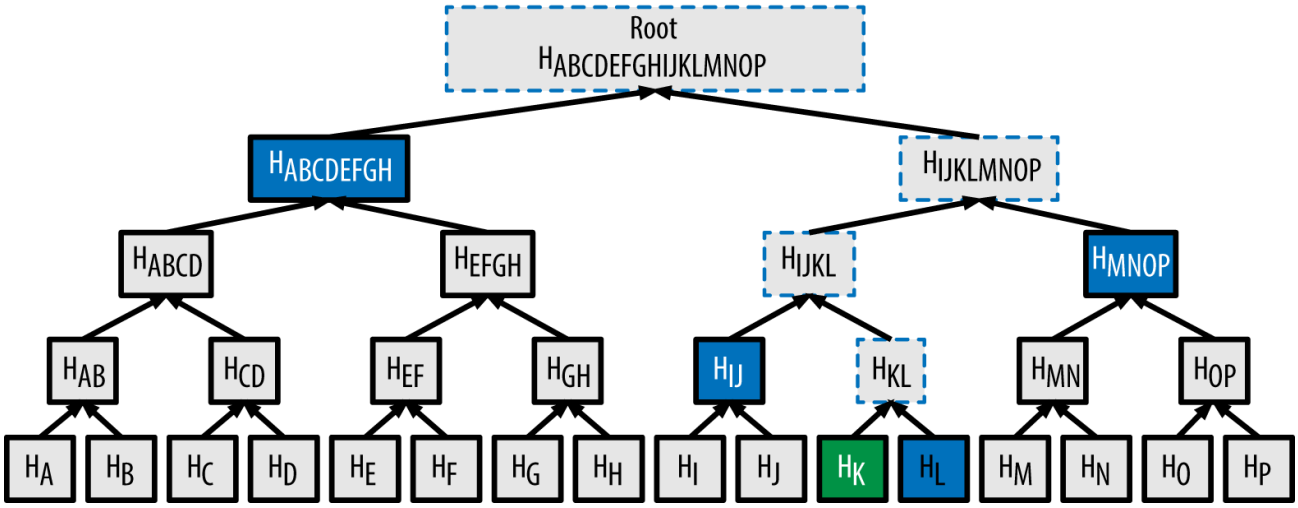
Παρακάτω παρουσιάζεται ένα παράδειγμα:

Για να επαληθεύσουμε τα δεδομένα [K], βρίσκοντας τη ρίζα του merkle tree, χρησιμοποιούμε μια κρυπτογραφική συνάρτηση για να παράγουμε το H(K). Προκειμένου να επικυρωθούν τα δεδομένα K, δεν χρειάζεται να αποκαλυφθεί το K.

H(K) όταν ενώνεται και περνάει στην κρυπτογραφική συνάρτηση μαζί με το υποσύνολο L, παράγεται το H(KL) αν δοθεί σαν εισοδο στην κρυπτογραφικη συνάρτηση το hash του H(KL) και του H(IJ) παράγει το H(IJKL). αν δοθεί σαν εισοδο στην κρυπτογραφικη συνάρτηση το hash του H(IJKL) και του H(MNOP) οδηγούν στο H(IJKLMNOP) αν δοθεί σαν εισοδο στην κρυπτογραφικη συνάρτηση

H(IJKLMNOP) και το H(ABCDEFGH) παράγεται το H(ABCDEFHGIJKLMNOP) το οποίο είναι η ρίζα του Merkle Tree.

Οπότε αποδείξαμε ότι το K είναι όντως σύνολο του δέντρου χωρίς να αποκαλύψουμε τα δεδομένα του [18].



Σχήμα 3.1: Merkle Tree

Κεφάλαιο 4

Decentralized Δίκτυα

Μια decentralized αρχιτεκτονική δικτύου κατανέμει φόρτο εργασίας σε διάφορα μηχανήματα, αντί να βασίζεται σε έναν κεντρικό διακομιστή. Αντίθετα, ένα κεντρικό δίκτυο βασίζεται σε έναν διακομιστή ως το κέντρο διανομής του. Αυτό το γεγονός μπορεί να οδηγήσει σε έλλειψη ασφάλειας, ζητήματα κόστους και σταθερότητας. Η λύση είναι να επεκταθεί το δίκτυο σε πολλές, απομακρυσμένες τοποθεσίες όπου όλες οι συναλλαγές γίνονται από διαφορετικούς διακομιστές, καθώς η διανομή δεδομένων σε διάφορους υπολογιστές επιτρέποντας αξιόπιστη λειτουργία. Ουσιαστικά, είναι ένας διασυνδεδεμένος ιστός υπολογιστών που παρέχει όλες τις δυνατότητες με τη μορφή αποθήκευσης ή υπολογιστικής ισχύος για την εκτέλεση ενός κοινού στόχου. Η επέκταση ενός δικτύου σε διάφορες τοποθεσίες μειώνει την αναγκαιότητα ενός κεντρικού διακομιστή, το οποίο βελτιώνει την ασφάλεια και τη σταθερότητα με το πρόσθετο μπόνους της εξοικονόμησης κόστους [14].

4.1 Blockchain

Το Blockchain είναι ένα decentralized, κατανεμημένο σύστημα, το οποίο αποτελείται από εγγραφές, οι οποίες ονομάζονται Blocks. Τα Blocks χρησιμοποιούνται για την αποθήκευση των συναλλαγών που πραγματοποιούνται στο δίκτυο, με τέτοιο τρόπο ώστε να μην γίνεται να αλλάξουν οι πληροφορίες που περιέχει, χωρίς να αλλαχθούν και όλα τα ακόλουθα Block. Τα Block περιέχουν πληροφορίες όπως η ημερομηνία, οι συμμετέχοντες και το κόστος κάθε συναλλαγής. Περιέχουν επίσης ένα μοναδικό κωδικό, το ”hash”, το οποίο είναι το αναγνωριστικό κάθε Block. Τα hashes παράγονται χρησιμοποιώντας κρυπτογραφικές συναρτήσεις.

Όταν ένα Block αποθηκεύσει πληροφορίες προστίθεται στο Blockchain, που είναι μια αλυσίδα από Blocks. Για να προστεθεί ένα Block στο Blockchain χρειάζεται:

• Πρέπει να πραγματοποιηθεί μια συναλλαγή. Σε πολλές περιπτώσεις, ένα μπλοκ ομαδοποιεί πολλές συναλλαγές.

• Αυτή η συναλλαγή πρέπει να επαληθευτεί. Αφού πραγματοποιηθεί μια αγορά, η συναλλαγή πρέπει να επαληθευτεί. Για το λόγο αυτό, ένα δίκτυο υπολογιστών ελέγχει ότι η συναλλαγή συνέβη με τον τρόπο που αναφέρεται. Δηλαδή, επιβεβαιώνουν τις λεπτομέρειες της αγοράς, συμπεριλαμβανομένου του χρόνου συναλλαγής, του ποσού των χρημάτων που καταναλώθηκαν και των συμμετεχόντων.

• Αυτή η συναλλαγή πρέπει να αποθηκευτεί σε ένα μπλοκ. Μετά την επαλήθευση της συναλλαγής σας ως ακριβής, το χρηματικό ποσό της συναλλαγής, η ψηφιακή σας υπογραφή και η ψηφιακή υπογραφή του άλλου συμμετέχοντα αποθηκεύονται σε ένα Block.

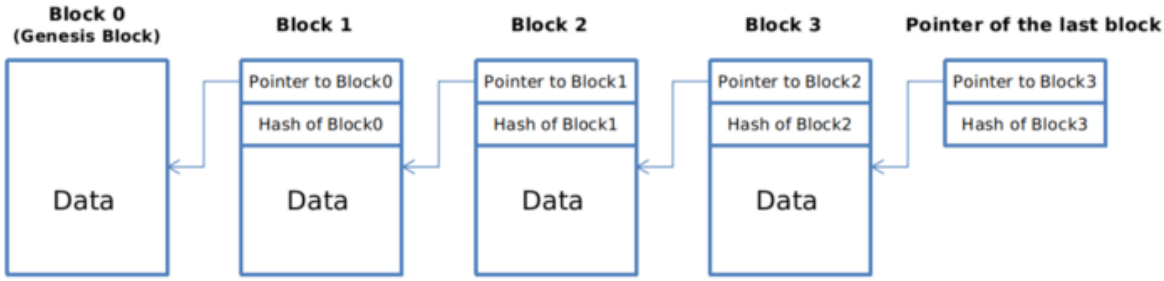
• Σε αυτό το μπλοκ πρέπει να περιέχεται το hash του. Μόλις επαληθευτούν όλες οι συναλλαγές ενός μπλοκ, πρέπει να έχει έναν μοναδικό κωδικό αναγνώρισης που ονομάζεται hash. Στο μπλοκ δίνεται επίσης το hash του πιο πρόσφατου μπλοκ που προστέθηκε στο blockchain. Μόλις υπολογιστεί το hash του, το μπλοκ μπορεί να προστεθεί στο blockchain. Όταν προστίθεται αυτό το νέο μπλοκ στο blockchain, καθίσταται διαθέσιμο σε όλους.

Η τεχνολογία blockchain προνοεί για τα ζητήματα ασφάλειας και εμπιστοσύνης με διάφορους τρόπους. Πρώτον, τα νέα μπλοκ αποθηκεύονται πάντα γραμμικά και χρονολογικά. Δηλαδή, προστίθενται πάντα στο «τέλος» του blockchain.

Αφού προστεθεί ένα Block στο τέλος του Blockchain, είναι πολύ δύσκολο να αλλαχθεί το πε-

ριεχόμενο του. Αυτό συμβαίνει επειδή κάθε Block περιέχει το δικό του hash, μαζί με το hash του προηγούμενου Block. Τα hashes δημιουργούνται από μια μαθηματική συνάρτηση που μετατρέπει τις ψηφιακές πληροφορίες σε σειρά αριθμών και γραμμάτων. Εάν αυτές οι πληροφορίες έχουν επεξεργαστεί με οποιονδήποτε τρόπο, αλλάζει και το εκάστοτε hash. Τα Block συνδέονται με Hash Pointers. Ο Hash Pointer αποτελείται από δύο μέρη:

- Δείκτης όπου αποθηκεύονται κάποιες πληροφορίες
- Το hash αυτών των πληροφοριών

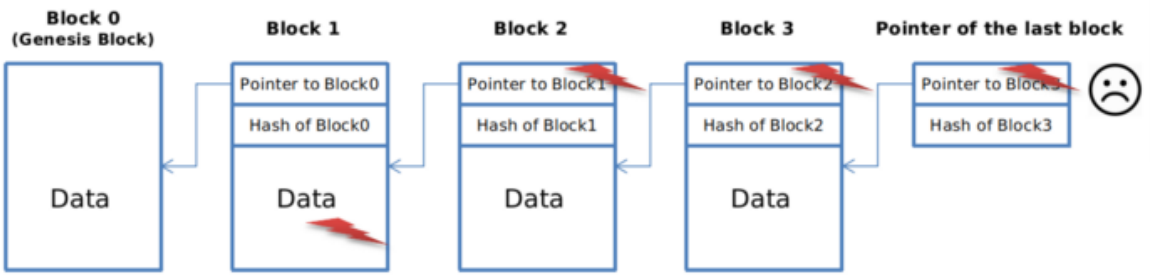


Σχήμα 4.1: Blockchain
https://medium.com/@zhaohuabing/hash-pointers-and-data-structures-f85d5fe91659

Ο δείκτης μπορεί να χρησιμοποιηθεί για τη λήψη των πληροφοριών, το hash μπορεί να χρησιμοποιηθεί για την επαλήθευση τους (την αλλαγή του περιεχομένου του Block). Πρέπει να σημειώσουμε ότι το hash που είναι αποθηκευμένο στο Hash Pointer είναι το hash ολόκληρων των δεδομένων του προηγούμενου Block, το οποίο περιλαμβάνει επίσης το Hash Pointer στο Block πριν από αυτόν. Αυτό καθιστά αδύνατο να παραβιαστεί ένα Block στο Blockchain χωρίς να γίνει αντιληπτό.

Παράδειγμα

- Ένας εισβολέας θέλει να παραβιάσει ένα Block της αλυσίδας, ας πούμε, Block 1.
- Ο εισβολέας άλλαξε το περιεχόμενο του Block 1
- Για να μην παρατηρηθεί η αλλαγή, πρέπει επίσης να αλλάξει το Hash pointer αυτού του Block στο επόμενο Block, το οποίο είναι το Block 2.
- Τώρα το περιεχόμενο του μπλοκ 2 έχει αλλάξει, οπότε πρέπει να αλλάξει και το Hash pointer του Block 3.
- Τέλος, ο εισβολέας πηγαίνει στο Hash Pointer του τελευταίου Block του Blockchain, το οποίο είναι αποκλεισμένο για αυτόν, επειδή διατηρούμε και θυμόμαστε το Hash Pointer.



Σχήμα 4.2: Αλλαγμένο Block
https://medium.com/@zhaohuabing/hash-pointers-and-data-structures-f85d5fe91659

Για την αντιμετώπιση του ζητήματος της εμπιστοσύνης, τα δίκτυα Blockchain έχουν εφαρμόσει δοκιμές για υπολογιστές που θέλουν να συμμετάσχουν και να προσθέσουν μπλοκ στην αλυσίδα. Οι δοκιμές, που ονομάζονται «μοντέλα συναίνεσης», απαιτούν από τους χρήστες να «αποδείξουν» τον εαυτό τους πριν μπορέσουν να συμμετάσχουν στο Blockchain. Ένα από τα πιο συνηθισμένα παραδείγματα ονομάζεται «απόδειξη της εργασίας», Power Of Work.

Στην απόδειξη του συστήματος εργασίας, οι υπολογιστές πρέπει να «αποδείξουν» ότι έχουν κάνει «δουλειά» επιλύοντας ένα περίπλοκο υπολογιστικό μαθηματικό πρόβλημα. Εάν ένας υπολογιστής επιλύσει ένα από αυτά τα προβλήματα, επιλέγονται για να προσθέσουν ένα Block στο Blockchain. Όμως, η διαδικασία προσθήκης Block στο Blockchain (mining) δεν είναι εύκολο.

Εάν ένας χάκερ ήθελε να συντονίσει μια επίθεση στο Blockchain, θα χρειαζόταν να ελέγξει περισσότερο από το 50% της συνολικής υπολογιστικής δύναμης στο Blockchain έτσι ώστε να είναι σε θέση να κατακλύσει όλους τους άλλους συμμετέχοντες του δικτύου [13, 15].

Κεφάλαιο 5

Η Αρχιτεκτονική του Συστήματος

5.1 Γράφοντας το Αρχείο

Όπως αναφέρθηκε παραπάνω, ένα δέντρο Merkle είναι ένα δέντρο στο οποίο κάθε κόμβος-φύλλο φέρει ετικέτα με το κρυπτογραφικό hash ενός μπλοκ δεδομένων και κάθε κόμβος οχι-φύλλο φέρει ετικέτα με το κρυπτογραφικό hash των ετικετών των θυγατρικών κόμβων του. Τα Hash δέντρα επιτρέπουν την αποτελεσματική και ασφαλή επαλήθευση του περιεχομένου των μεγάλων δομών δεδομένων.

Όταν ένας πελάτης γράφει δεδομένα, ένα DFSOutputStream αποθηκεύει τα δεδομένα σε ένα προσωρινό τοπικό αρχείο. Προσθέσαμε ένα buffer που επίσης συσσωρεύει αυτά τα δεδομένα και όταν αποθηκεύει δεδομένα ίσο με μέγεθος μπλοκ HDFS (128 MB από προεπιλογή), αυτός ο buffer ανακατευθύνεται σε μια κλάση με το όνομα MakeMerkleTree.java. Τα δεδομένα χωρίζονται σε τεμάχια ίσα με το μέγεθος Merkle Chunk (όπως θα δούμε παρακάτω επιλέξαμε να είναι 10000 bytes). Κάθε Merkle Chunk κατακερματίζεται σύμφωνα με την κρυπτογραφικό αλγόριθμο SHA256, κάθε hash είναι ένα φύλλο του δέντρου που ονομάζεται απόδειξη(proof). Στη συνέχεια, το δέντρο δημιουργείται ξεκινώντας από τα φύλλα, αναδρομικά. Όπως παρατηρούμε στην παρακάτω λειτουργία στην κλάση MerkleTree.java:

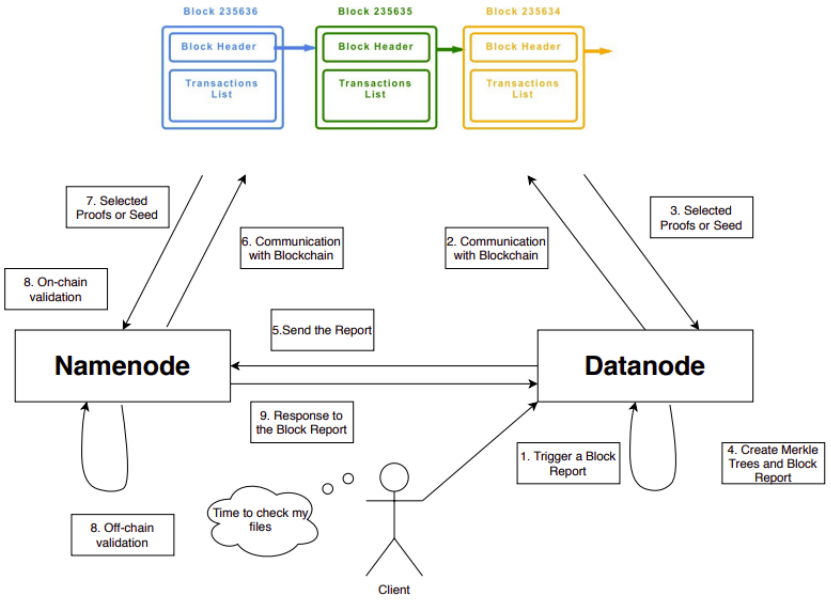
Μόλις δημιουργηθεί το δέντρο, ο Datastreamer.java ορίζει στο τρέχον μπλοκ (η κλάση μπλοκ αποτελείται από πληροφορίες σχετικά με το μπλοκ που στέλνονται στο Namenode, δεν περιλαμβάνει τα δεδομένα, περιλαμβάνει το blockid - ένα αναγνωριστικό μοναδικό για κάθε μπλοκ, τη σφραγίδα δημιουργίας του μπλοκ και τα bytes) τα πεδία roothashand myproof, που είναι ο κατακερματισμός της ρίζας του δέντρου και μια λίστα που αποτελείται από όλα τα hashes των φύλλων. Στη συνέχεια, χρησιμοποιώντας τη μέθοδο addBlock στέλνετε αυτό το Block στο Namenode. Τα δεδομένα αποστέλλονται χρησιμοποιώντας buffer πρωτοκόλλου, οι οποίοι αποτελούν μηχανισμό για σειριοποίηση δεδομένων - σκεφτείτε XML, αλλά μικρότεροι, γρηγορότεροι και απλούστεροι.

Ο Namenode λαμβάνει τα δεδομένα και αποθηκεύει τη λίστα των αποδείξεων σε ένα Rockdb. Το RocksDB είναι μια ενσωματωμένη βάση δεδομένων υψηλής απόδοσης για δεδομένα κλειδιού-τιμής. Το κλειδί αποτελείται από το αναγνωριστικό του μπλοκ και το αναγνωριστικό της απόδειξης, που είναι το index του στη λίστα, οπότε τόσο η τιμή όσο και το κλειδί είναι Strings. Το rootHash αποθηκεύεται στη μνήμη του Namenode και στο FSimage, ώστε να μπορεί να ανακτηθεί κατά την έναρξη του HDFS. Η διαδικασία αποστολής των δεδομένων στο pipeline των Datanodes παραμένει η ίδια, τίποτα δεν έχει αλλάξει. Οι Datanodes δεν γνωρίζουν τίποτα για τις αποδείξεις και το rootHash. Μέχρι στιγμής έχουμε αναλύσει τη διαδικασία στο Client Side, κατά τη διάρκεια της εντολής put του αρχείου. Έτσι, ο Namenode, έχει αποθηκεύσει τις πληροφορίες σχετικά με το rootHash και τις αποδείξεις. Αυτό ήταν το πρώτο μέρος του συστήματος σχετικά με τα Merkle Trees.

5.2 Block Report

Ένα Block Report πραγματοποιείται κάθε 6 ώρες, αλλά μπορούμε να ενεργοποιήσουμε ένα BlockReport με την εντολή hdfs dfsadmin -triggerBlockReport ip: port. Το BlockReport ξεκινάει σαν διαδικασία πάντα από τον Datanode, καθώς όπως αναφέραμε ξανά ο Namenode δεν επιδιώκει ποτέ επικοινωνία

με τους Datanodes. Ενισχύσαμε τη λειτουργία BlockReport χρησιμοποιώντας το Blockchain και τις Merkle αποδείξεις που αναφέρθηκαν παραπάνω.



Σχήμα 5.1: Η Αρχιτεκτονική του Block Report

Η κύρια ιδέα είναι ότι κάθε φορά που ενεργοποιείται ένα BlockReport, ο Datanode θα ενημερώνεται για την ανάδειξη ορισμένων αποδείξεων Merkle. Για κάθε μπλοκ που ο Datanode έχει αποθηκευμένο πρέπει να στείλει έναν συγκεκριμένο αριθμό αποδείξεων, το οποίο ζητείται από «κάποιον». Η πρώτη ιδέα ήταν το Namenode να είναι ”κάποιος” που καθορίζει το πλήθος και την ταυτότητα των αποδείξεων που πρέπει να δώσει ο Datanode, αλλά σε περίπτωση που το Namenode είναι κακόβουλος και συνεργάζεται με τον Datanode δεν μπορεί να εφαρμοστεί αυτή η μέθοδος. Αυτοί οι κόμβοι μπορούν να εξαπατήσουν προς όφελός τους, χωρίς να γίνουν αντιληπτοί. Για παράδειγμα, ο Namenode μπορεί να αποφασίσει να ζητά τις ίδιες αποδείξεις κάθε φορά, επειδή γνωρίζει ότι τα δεδομένα αυτών των αποδείξεων δεν είναι κατεστραμμένα και να αφήσει το Datanode να διαγράψει ή να καταστρέψει όλα τα υπόλοιπα δεδομένα. Έτσι, με βάση την υπόθεση ότι κάθε κόμβος μπορεί να είναι κακόβουλος, πήραμε την απόφαση να αφήσουμε τη διαδικασία καθορισμού σε ένα ιδιωτικό Blockchain που είναι ένα αξιόπιστο δίκτυο, μέσω της εκτέλεσης smart contracts που γράψαμε, θα αναλύσουμε τον κώδικα και τη λειτουργία τους παρακάτω.

Τώρα που μιλήσαμε για τη διαδικασία καθορισμού των αποδείξεων που πρέπει να δοθούν, ας ρίξουμε μια ματιά στη διαδικασία αποστολής και επικύρωσης των Merkle Proofs. Σκεφτείτε ότι το blockchain επιστρέφει στον Datanode μια λίστα αριθμών (θα αναλύσουμε την παρακάτω διαδικασία, αλλά ας το θεωρήσουμε δεδομένο ότι το Blockchain επιστρέφει μια λίστα αριθμών στο Datanode). Το Datanode παίρνει αυτήν τη λίστα και καθορίζει ποιοι αριθμοί αντιστοιχούν σε ποιο μπλοκ (αυτό εξαρτάται από τη διαδικασία απόδειξης Merkle που αποφασίσαμε να χρησιμοποιήσουμε). Έτσι, για παράδειγμα, έχουμε τη λίστα [0,1,3,1,2,6,3], ο Datanode γνωρίζει τώρα ότι για το πρώτο Block πρέπει να παρέχει τις αποδείξεις 0,1,3 για το δεύτερο Block 1,2,6 και για τον τρίτο την απόδειξη 3. Τότε για κάθε Block για το οποίο ο Datanode πρέπει να στείλει απόδειξη δημιουργεί το Merkle Tree του, αφού οι Datanodes έχουν στο δίσκο τα πραγματικά δεδομένα του αρχείου. Η δημιουργία του Merkle Tree εξηγείται παραπάνω. Δεδομένου ότι ζητούνται αποδείξεις Merkle για σχεδόν κάθε μπλοκ που είναι αποθηκευμένο στο Datanode, χρησιμοποιήσαμε νήματα για να παραλληλίσουμε τη διαδικασία και να βελτιώσουμε την απόδοση, κάθε διαθέσιμο νήμα δημιουργεί ένα Merkle Tree ενός διαφορετικού μπλοκ. Οι αριθμοί αντιστοιχούν στο index των αποδείξεων του Merkle Tree (οι αποδείξεις είναι

τα φύλλα του). Δεδομένου ότι το δέντρο δημιουργήθηκε και εντοπίστηκαν οι αποδείξεις που πρέπει να δώσει ο Datanode, είναι καιρός να παράγουμε τις απαραίτητες πληροφορίες για τη διαδικασία επικύρωσης. Η διαδικασία εξηγείται στην ενότητα Merle Trees. Μόλις δημιουργηθεί η διαδρομή, δημιουργείται ένα αντικείμενο MerkleRequest. Αυτό το αντικείμενο είναι μέρος του BlockReport και περιέχει, τις διαδρομές που δημιουργήσαμε, μια λίστα που καθορίζει εάν η αντίστοιχη καταχώρηση στη λίστα των διαδρομών είναι το δεξί ή το αριστερό παιδί, τα αναγνωριστικά όλων των μπλοκ που συμμετέχουν στο MerkleRequest, τα δεδομένα (τα 10000 bytes από τα οποία το φύλλο δημιουργήθηκε, αυτό καθιστά τη διαδικασία ακόμη πιο ασφαλή, καθώς ο Datanode αποδεικνύει ότι περιέχει επίσης τα δεδομένα).

Στη συνέχεια, το BlockReport αποστέλλεται στο Namenode, ο οποίος πρέπει να επικυρώσει τα αποτελέσματα. Καθώς ο Namenode λαμβάνει το BlockReport, συνδέεται με το Blockchain και ανακτά τα αποτελέσματα της λίστας αριθμών που το Blockchain επέστρεψε στο Datanode νωρίτερα. Αυτό επιτυγχάνεται μέσω events που εκπέμπει το Smart Contract. Τα events είναι σήματα που μπορούν να ενεργοποιηθούν από τα smart contracts, που περιλαμβάνουν πληροφορίες για την εκτέλεση ή τα δεδομένα ενός smart contract. Υπάρχουν δύο τρόποι επικύρωσης των αποτελεσμάτων. Επειδή δεν είναι αξιόπιστος ο Namenode, εφαρμόσαμε μια επικύρωση On-chain και μια τοπική επικύρωση. Η on-chain επικύρωση είναι πιο ασφαλής, καθώς το αποτέλεσμα του smart contract μπορεί πάντα να είναι αξιόπιστο, αντίθετα με αυτό που παρέχεται από την τοπική επικύρωση στο Namenode. Ο Namenode μπορεί να καταστρέψει τα αποτελέσματα και να παρέχει μη έγκυρες πληροφορίες στο σύστημα. Από την άλλη πλευρά, η εκτέλεση του smart contract αποθηκεύεται στο Blockchain και ο καθένας μπορεί να το διαβάσει και να επιβεβαιώσει την ενέργεια του Namenode ανά πάσα στιγμή. Ο αλγόριθμος είναι ο ίδιος για τις δύο περιπτώσεις, οπότε θα αναλύσουμε στο επόμενο τμήμα την επικύρωση On-chain, οπότε ας επικεντρωθούμε στην τοπική. Ας ρίξουμε μια ματιά στις κύριες λειτουργίες που εκτελεί η διαδικασία επικύρωσης:

```
for every block:
   for every proof:
      root = findtheroot(path, parents, proof);
      if(root != storedRoot and hash(data) == SoredProofInDatabase):
         invalid.add(block);
```

Σχήμα 5.2: Πρώτο Βήμα στην Διαδικασία Επικύρωσης

Αρχικά, για κάθε block ελέγχονται όλα τα proof που έχουν ζητηθεί, καλείται η συνάρτηση findtheroot, η οποία βρίσκει τη ρίζα του δέντρου από τις διαδρομές που έχουν σταλεί από τον Datanode, όπως εξηγείται στην ενότητα Merkle Trees, η συνάρτηση παρουσιάζεται παρακάτω:

```
function findtheroot(path, parents, success):
   while(paths not empty):
      if(parents(0) == 1):
         success = hash(success, paths(0));
      else if(parents(0) == 0):
         success = hash(paths(0), success));
      remove(paths(0), parents(0));
return success;
```

Σχήμα 5.3: Εύρεση Ρίζας

Η παραπάνω συνάρτηση επιστρέφει τη ρίζα που βρέθηκε από αυτήν τη διαδικασία, παρατηρούμε πως η λίστα parents περιέχει είτε τον αριθμό 1 είτε 0 για να γνωρίζουμε με ποιά σειρά πρέπει να κατακερματιστούν οι δύο ακολουθίες. Τέλος, η πρώτη συνάρτηση που αναφέραμε ελέγχει εάν η ρίζα που βρέθηκε από αυτήν τη διαδικασία είναι ίδια με αυτήν που είναι αποθηκευμένη στη μνήμη του Namenode. Επίσης, ελέγχει εάν το hash των δεδομένων που στάθηκαν απο το Datanode (τα δεδομένα από τα οποία προκύπτει το εκάστοτε proof) είναι το ίδιο με το hash που είναι αποθηκευμένο στη βάση δεδομένων Rockdb. Εάν δεν είναι τα ίδια, το μπλοκ προστίθεται στα μη έγκυρα μπλοκ και ζητείται από το Datanode να το διαγράψει. Αυτή είναι η περίπτωση που το Namenode είναι αξιόπιστος. Μπορούμε να καταλάβουμε ότι ένας κακόβουλος Namenode μπορεί να αλλάξει τα αποτελέσματα, καθώς όλα εκτελούνται στον επεξεργαστή του. Γι ’αυτό είναι σημαντικό να εκτελέσουμε την επικύρωση on-chain εάν θέλουμε το σύστημά μας να είναι 100% ασφαλές.

5.3 Smart Contracts

Τα smart contracts είναι απλά προγράμματα υπολογιστών. Μόλις υποβληθεί το smart contract στο Blockchain, ο κώδικας δεν μπορεί να αλλάξει. Το αποτέλεσμα της εκτέλεσης ενός smart contract είναι το ίδιο για όλους όσους το εκτελούν. Υπάρχουν 2 λόγοι για να επικοινωνήσουμε με το Blockchain. Ο πρώτος είναι για τον προσδιορισμό των αποδείξεων που ο Datanode θα στείλει στον Namenode και ο δεύτερος για την επικύρωση τους. Αρχικά ας εξηγήσουμε πώς γίνεται ο προσδιορισμός των αποδείξεων. Ο προσδιορισμός των αποδείξεων εξαρτάται από τη διαδικασία επικύρωσης που αποφασίσαμε να χρησιμοποιήσουμε. Μπορούμε να επιλέξουμε πάνω από τρεις επιλογές. Η πρώτη είναι μία απόδειξη ανά μπλοκ, η δεύτερη είναι K αποδείξεις ανά μπλοκ και η τρίτη είναι N αποδείξεις από όλα τα μπλοκ. Ας εξετάσουμε το πρώτο. Εξετάζουμε όλα τα μπλοκ και κάνουμε μια λίστα με αριθμούς που περιέχει πόσες πιθανές αποδείξεις έχει κάθε μπλοκ. Ύστερα επικοινωνούμε με το Ethereum:

```
contract Numbers:

  function number(data, time):
    for(i=0 until i<data.length):
      result.add(hash(data(i), i, time) mod data(i));
    emit event NumberIssue(time, result, senderAddress);

  function numberK(data, time, k):
    for(i=0 until i<data.length):
      for(j=0; until j<k):
        result.add(hash(data(i), time, j, i, k) mod data(i));
    emit event NumberIssue(time, result, senderAddress);

  function kproofs(data, time, k):
    total = 0;
    for(i=0 until i<data.length):
      total  = total + data(i);
    step = total div k;
    begin = total mod k;
    for(i=begin until i<total; i=i+step):
      result.add(hash(total, time, i) mode total);
    emit event NumberIssue(time, result, senderAddress);
```

Σχήμα 5.4: Smart Contract

Η πρώτη μέθοδος αντιστοιχεί στην πρώτη επιλογή. Παίρνει ως παράμετρο μια λίστα αριθμών των αποδείξεων που υπάρχουν σε κάθε μπλοκ και το χρόνο που επιτεύχθηκε η επικοινωνία με το Ethereum. Έπειτα πραγματοποιείται κατακερματισμός μεταξύ των δεδομένων, του index στη λίστα και του χρόνου. Ο αριθμός που βρέθηκε κανονικοποιείται μέσω της λειτουργίας modulo με τα δεδομένα. Θέλουμε το αποτέλεσμα να είναι μεταξύ του μηδέν και του data[i].

Η δεύτερη μέθοδος είναι η ίδια με την πρώτη με τη προσθήκη μιας επανάληψης for εφόσον ζητάμε κ αποδείξεις από κάθε μπλοκ.

Η τρίτη επιλογή είναι λίγο πιο περίπλοκη. Η ιδέα είναι να λάβουμε τυχαίες αποδείξεις από όλα τα μπλοκ. Στην αρχή, λοιπόν, αθροίζουμε όλες τις τιμές των αριθμών των αποδείξεων που υπάρχουν σε κάθε μπλοκ. Ως αποτέλεσμα, λαμβάνουμε τον αριθμό όλων των υπαρχόντων - δυνητικά ζητούμενων - αποδείξεων στον Datanode. Δεδομένου ότι θέλουμε n από αυτά, κανονικοποιούμε τη διαδικασία με τη χρήση λειτουργιών διαίρεσης και modulo. Τώρα ο κατακερματισμός πραγματοποιείται μεταξύ του συνολικού αριθμού των αποδείξεων, του χρόνου και του index της ζητούμενης απόδειξης (0-n) modulo της συνολικής τιμής προκειμένου να κανονικοποιηθεί η τιμή της ζητούμενης απόδειξης μεταξύ μηδέν και συνολικού αριθμού αποδείξεων. Το αποτέλεσμα επιστρέφεται στον Datanode ο οποίος καθορίζει ποια απόδειξη αντιστοιχεί σε ποιο μπλοκ. Στη συνέχεια, η διαδικασία του BlockReport είναι αυτή που αναφέρεται παραπάνω.

Ένας άλλος τρόπος προσδιορισμού των αποδείξεων που πρέπει να δώσει ο Datanode είναι μέσω ενός αριθμού που ονομάζεται seed. Το seed είναι ένας τυχαίος αριθμός που είναι το σημείο εκκίνησης για τη δημιουργία μιας τυχαίας ακολουθίας αριθμών. Η ιδέα είναι ότι το Blockchain αντί να στέλνει μια ολόκληρη λίστα αριθμών και να καταναλώνει χρόνο για τη μεταφορά μεγάλου όγκου δεδομένων, να παρέχει έναν αριθμό που ονομάζεται seed. Ο Datanode παίρνει το seed και μέσω μιας ντετερμινιστικής διαδικασίας δημιουργεί μια ακολουθία αριθμών στο εύρος [0, total], το total είναι οι συνολικές αποδείξεις που διαθέτει ένα Datanode. Παρατίθεται το smart contract που χρησιμοποιήσαμε για τη δημιουργία του seed και ο κώδικας για τη δημιουργία της ακολουθίας αριθμών.

```
contract Seed:

  function seed(time):
     s = hash(time) mod 2147483647
     emit event Seeds(time, s);
```

Σχήμα 5.5: Seed

```
Random rand = new Random(seed);
for(i=0 until i<NumberofProofs):
   proofs.add(rand.nextNumber mod total);
```

Σχήμα 5.6: Δημιουργία Ακολουθίας

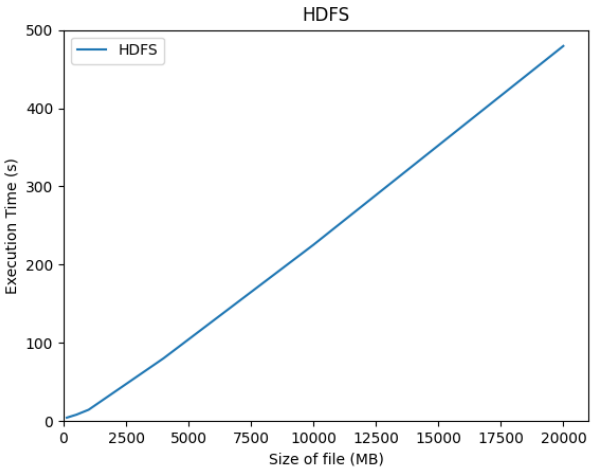
Ο δεύτερος λόγος που το πρόγραμμα επικοινωνεί με το Blockchain είναι η on-chain επικύρωση των αποδείξεων που παρέχονται. Ο Namenode έρχεται σε επαφή με το Blockchain και εκτελεί το αντίστοιχο smart contract. Η διαδικασία είναι η ίδια με την off-chain επικύρωση. Η εκτέλεση του smart contract επιστρέφει μια λίστα Boolean που περιέχει τα αποτελέσματα της επικύρωσης. Ο Namenode προσπελαύνει τα αποτελέσματα και στέλνει στον Datanode ποια μπλοκ δεν είναι έγκυρα. Το αποτέλεσμα της επικύρωσης είναι τώρα on-chain έτσι ώστε ο καθένας να μπορεί να ανακτήσει και να επικυρώσει τις ενέργειες του Namenode. Τώρα κάθε απόπειρα αλλοίωσης των αποτελεσμάτων να μπορεί να γίνει αντιληπτή.

Κεφάλαιο 6

Μετρήσεις και Συμπεράσματα

Η πειραματική διαδικασία πραγματοποιήθηκε στον server του εργαστηρίου. Χρησιμοποιήσαμε τους clones για να ρυθμίσουμε το σύστημα. Χρησιμοποιήσαμε ένα Namenode, τρείς Datanodes και ένα μηχάνημα για να τρέξουμε το ιδιωτικό Blockchain.

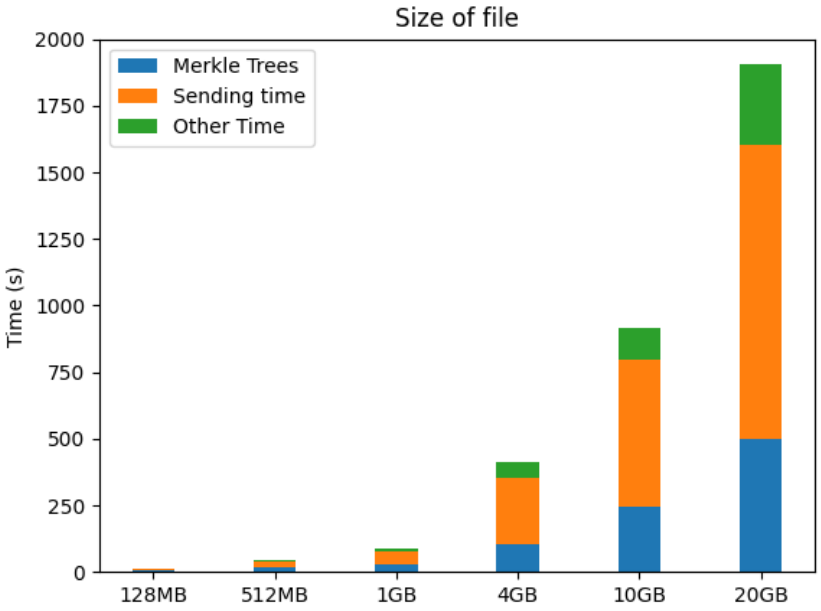
Το πρώτο σετ πειραμάτων αφορά τη HDFS (το κλασικό Hadoop χωρίς τις επεκτάσεις που προσθέσαμε). Η πειραματική διαδικασία περιλαμβάνει τη χρονομέτρηση της εντολής put κατά τη χρήση διαφορετικού μεγέθους αρχείων με συντελεστή αντιγραφής ίσο με δύο. Πραγματοποιήσαμε πέντε φορές κάθε πείραμα και υπολογίσαμε τη μέση τιμή. Τα αποτελέσματα παρουσιάζονται παρακάτω:



Σχήμα 6.1: Πειράματα του HDFS

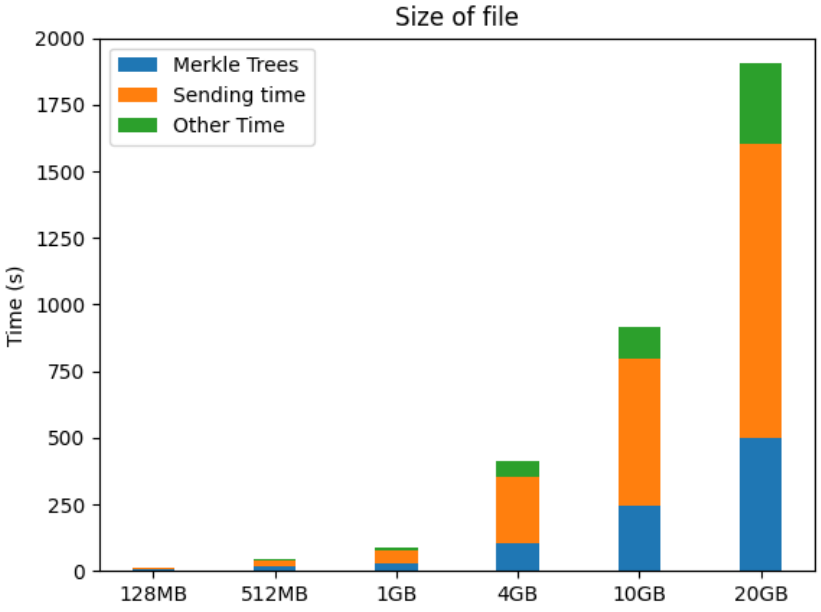
Στη συνέχεια, έγιναν μετρήσεις στο HDFS-Ethereum που περιλαμβάνει τα Merkle Trees. Όπως και πριν, η χρονομέτρηση πραγματοποιήθηκε στην εντολή put, μετρώντας τόσο τον συνολικό χρόνο όσο και τον μεμονωμένο χρόνο που χρειάζεται ο πελάτης για να δημιουργήσει το Merkle Tree για κάθε Block, καθώς και τον χρόνο που απαιτείται για την αποστολή των δεδομένων Merkle Tree στο Namenode (root hash και φύλλα). Η μέτρηση πραγματοποιήθηκε 5 φορές, οπότε για κάθε Block πήραμε τον μέσο όρο 5 μετρήσεων και στη συνεχεια προσθέσαμε το αποτέλεσμα που πήραμε για κάθε Block για τα μεγέθη που αναφέραμε παραπάνω, για να βρούμε το συνολικό τους χρόνο. Χρησιμοποιήθηκαν διαφορετικά μεγέθη Merkle Chunk. Επίσης, η εντολή δόθηκε από το Namenode, αργότερα θα εξετάσουμε την απόδοση εάν η εντολή δίνεται από διαφορετικό μηχάνημα. Τα αποτελέσματα παρουσιάζονται παρακάτω:

Για Merkle Chunk 512 bytes:



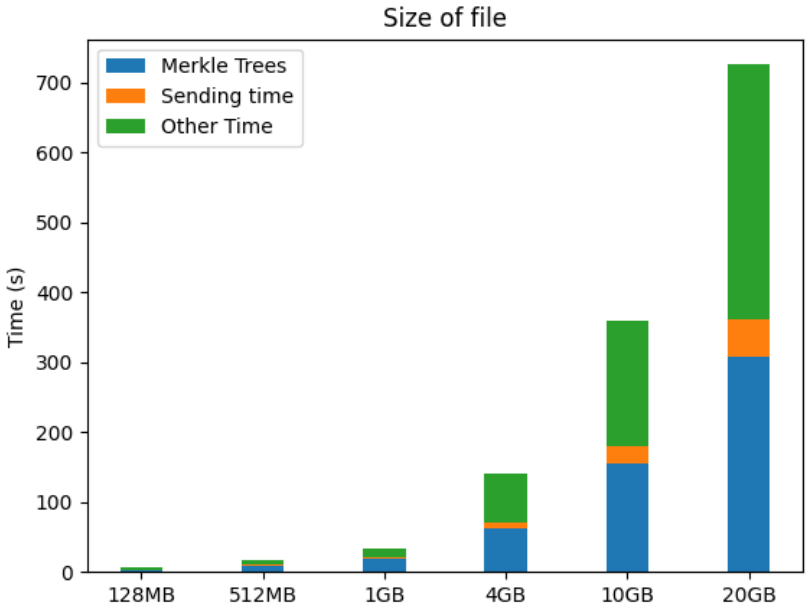
Σχήμα 6.2: Πειράματα για Merkle Chunk 512 bytes

Για Merkle Chunk 1024 bytes:



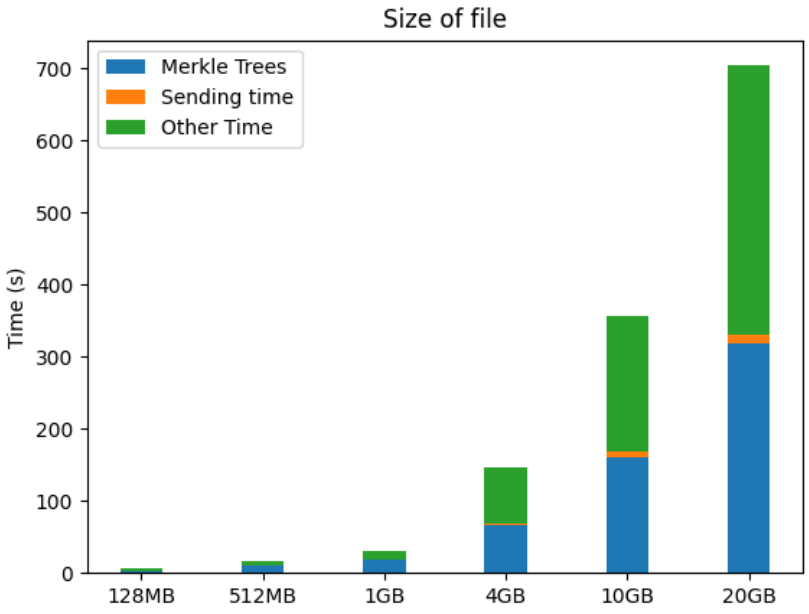
Σχήμα 6.3: Πειράματα για Merkle Chunk 1024 bytes

Για Merkle Chunk 10000 bytes:



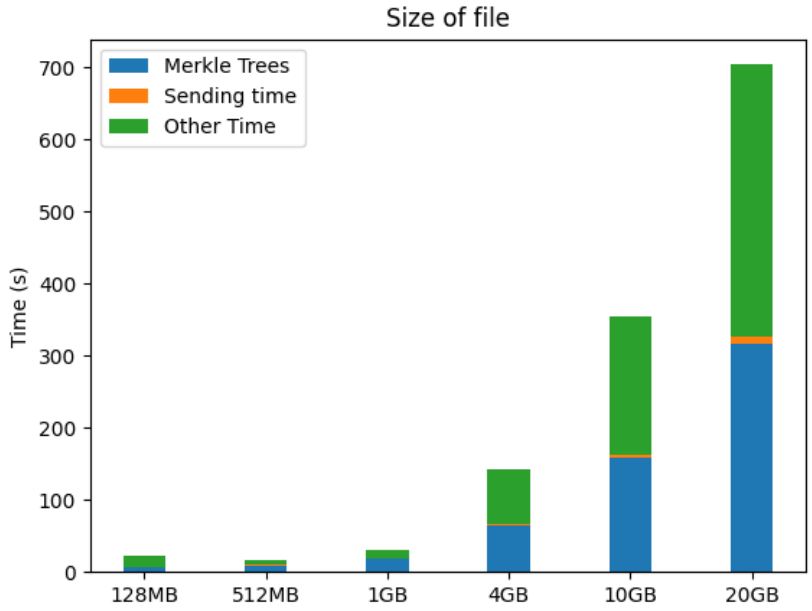
Σχήμα 6.4: Πειράματα για Merkle Chunk 10000 bytes

Για Merkle Chunk 100000 bytes:



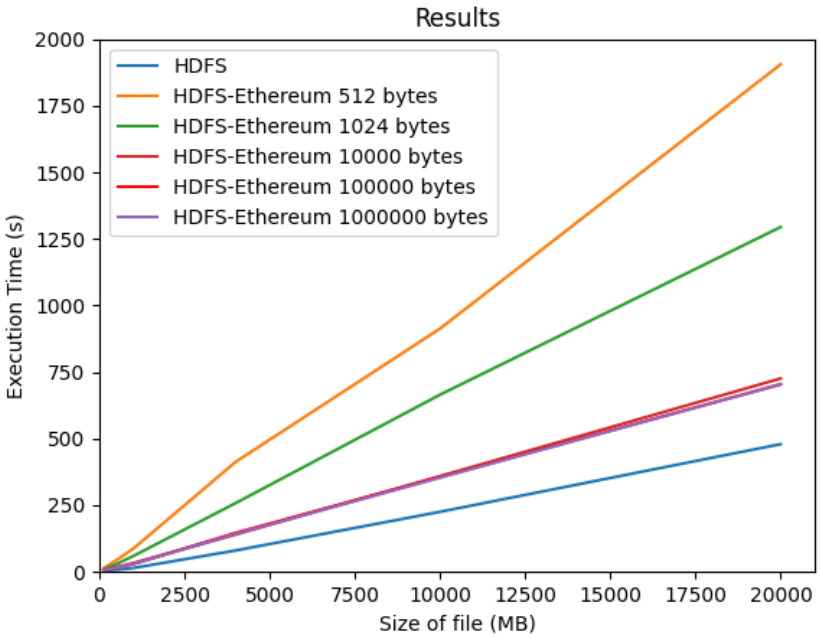
Σχήμα 6.5: Πειράματα για Merkle Chunk 100000 bytes

Για Merkle Chunk 1000000 bytes:



Σχήμα 6.6: Πειράματα για Merkle Chunk 1000000 bytes

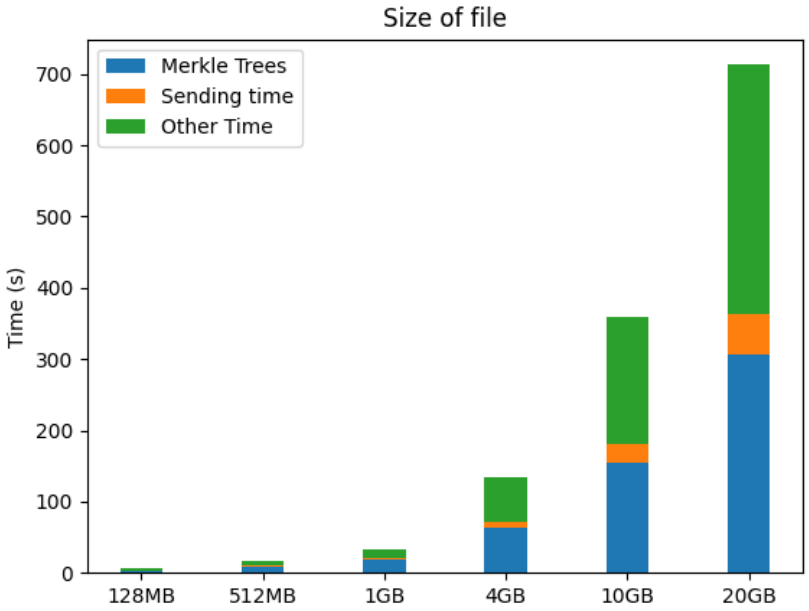
Παρακάτω παρουσιάζεται το διάγραμμα που περιέχει όλα τα πειράματα:



Σχήμα 6.7: Διάγραμμα με όλα τα Πειράματα

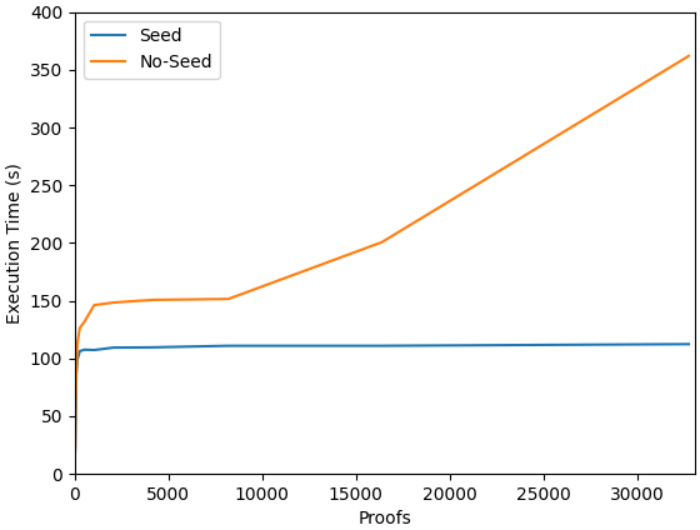
Παρατηρούμε πως για μέγεθος μεγαλύτερο των 10.000 bytes Merkle Chunk ο χρόνος που απαιτείται για την εκτέλεση της εντολής put είναι σχεδόν το ίδιος. Έτσι είναι πιο ασφαλές να έχουμε ένα μεγαλύτερο Merkle Tree, οπότε αποφασίσαμε ότι τα 10.000 bytes είναι το πιο αποτελεσματικό μέγεθος για να διατηρήσουμε την ισορροπία τόσο της απόδοσης όσο και της ασφάλειας του συστήματος.

Οι παραπάνω μετρήσεις πραγματοποιήθηκαν λαμβάνοντας υπόψη το ίδιο τον Namenode ως πελάτη. Χρησιμοποιώντας το 10.000 bytes merkle chunk, το οποίο θεωρήθηκε το πιο αποτελεσματικό, οι μετρήσεις πραγματοποιήθηκαν ξανά, θέτοντας έναν διαφορετικό κόμβο ως πελάτη. Τα αποτελέσματα παρουσιάζονται παρακάτω:

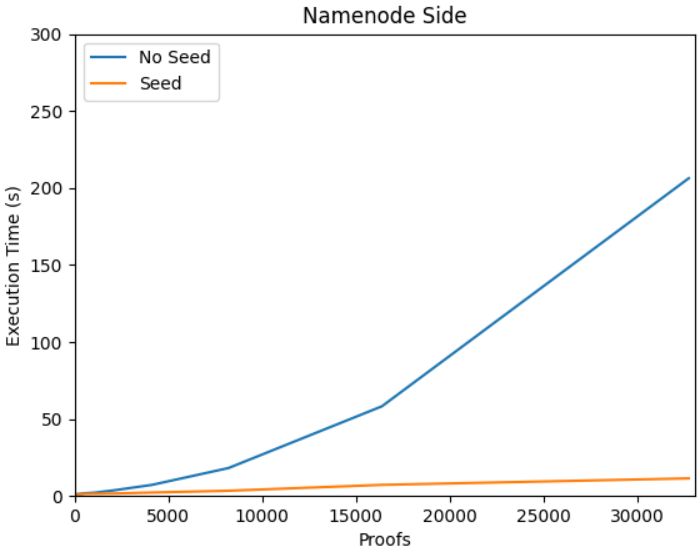


Σχήμα 6.8: Πειράματα για Διαφορετικό Πελάτη

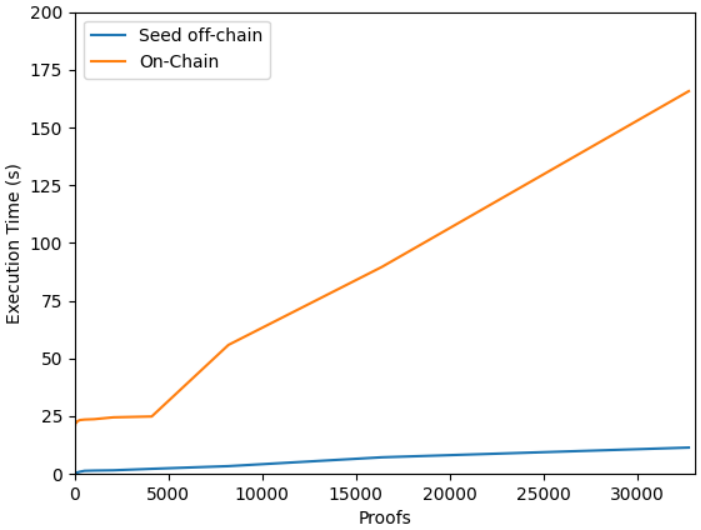
Στη συνέχεια πραγματοποιήθηκαν οι μετρήσεις Blockchain. Πιο συγκεκριμένα, μετρήσαμε την ώρα όπου ο Namenode χρειάζεται για να ανακτήσει τις πληροφορίες από το Blockchain και να επικυρώσουμε το BlockReport, ανάλογα με τον αριθμό των αποδείξεων που ζήτησε στον Datanode. Μετρήσαμε επίσης το χρόνο που χρειάζεται για να δημιουργήσει ο Datanode το BlockReport από το μηδέν, συμπεριλαμβανομένης της επικοινωνίας με το Blockchain. Ο προσδιορισμός των αποδείξεων γίνεται και είτε με τη χρήση seed, είτε το Blockchain επιστρέφει τη λίστα με τις αποδείξεις που θα ζητηθούν. Η επικύρωση μπορεί να γίνει είτε τοπικά είτε on-chain. Και οι δύο περιπτώσεις εξετάζονται παρακάτω:



Σχήμα 6.9: Datanode Seed vs no-Seed



Σχήμα 6.10: Namenode Seed vs no-Seed



Σχήμα 6.11: Namenode On-chain vs Off-chain

Από το σχήμα 6.9 παρατηρούμε σημαντική βελτίωση στους χρόνους με τη χρήση του Seed. Η μεταφορά και η καταγραφή μεγάλου όγκου δεδομένων είναι κατασταλτικός παράγοντας για τη λειτουργία του Blockchain. Με τη χρήση του Seed, όμως όπου το Smart Contract επιστρέφει μόνο ένα αριθμό η βελτίωση είναι σημαντική. Επίσης ο χρόνος παραμένει σχετικά σταθερός από έναν αριθμό αποδείξεων και μετά. Καθώς δημιουργεί τον ίδιο αριθμό Merkle Trees. Ο Datanode στον οποίο πραγματοποιήθηκαν τα πειράματα διέθετε 49 blocks. Παρατηρούμε οτι απο τον αριθμό αποδείξεων 256 και μετά ο Datanode δημιουργούσε 49 δέντρα τα οποία αντιστοιχούν σε όλα του τα block. Οπότε είναι λογικό ο χρόνος να είναι παρόμοιος, καθώς ο επεξεργαστικός όγκος είναι η δημιουργία των Merkle Trees από τα αρχεία.

Από το σχήμα 6.10 παρατηρούμε πως ο Namenode έχει την ίδια συμπεριφορά όσον αφορά το Seed. Με τη χρήση του Seed οι χρόνοι βελτιώνονται αισθητά, καθώς δεν υπάρχει αναμονή στο transaction με το Blockchain για τη μεταφορά μεγάλου όγκου δεδομένων. Το Smart Contract επιστρέφει έναν αριθμό και αναλαμβάνει ο Namenode να δημιουργήσει την ακολουθία των αριθμών.

Από το σχήμα 6.11 παρατηρούμε πως το On-chain validation είναι πιο χρονοβόρο και από τις δύο παραπάνω μεθόδους, γενικά αναμενόμενο. Το Blockchain θέλει περίπου 20 δευτερόλεπτα για να κάνει mine ένα block. Ο αριθμός των αποδείξεων που μπορούμε να στείλουμε σε κάθε transaction για επικύρωση είναι 128. Ο αριθμός των transaction που μπορούν να μπουν σε ένα block είναι συγκεκριμένος. Παρατηρούμε λοιπόν πως μέχρι τον αριθμό αποδείξεων 5.000 οι χρόνοι μας παραμένουν σταθεροί. Στη συνέχεια, ο χρόνος βλέπουμε πως ανεβαίνει ανάλογα με τα batches που στέλνουμε, αφού κάθε block χωράει μέχρι 32 batches των 128 αποδείξεων.

Για τον προσδιορισμό των αποδείξεων είναι προφανώς καλύτερη η μέθοδος με το seed, καθώς δεν δημιουργεί κάποιο ελάττωμα και βελτιώνει κατά πολύ το performance του προγράμματος. Όσον αφορά το validation, το On-chain είναι ο μόνος τρόπος να είμαστε σίγουροι για το αποτέλεσμα της διαδικασίας, αλλά είναι αρκετά χρονοβόρο και δεν βοηθάει καθόλου την απόδοση του προγράμματος μας. Από την άλλη μεριά το Off-chain προσφέρει μια γρήγορη επικύρωση, αλλά δημιουργεί κενά στην ασφάλεια, καθώς ο Namenode δεν ελέγχεται από κάπου, άρα πρέπει να θεωρηθεί αξιόπιστος. Το ποιά μέθοδο θα αποφασίσουμε να χρησιμοποιήσουμε βασίζεται στην πλατφόρμα που θέλουμε να χρησιμοποιήσουμε την παρούσα εφαρμογή. Στην περίπτωση που ο Namenode μπορεί να θεωρηθεί αξιόπιστος είναι σαφώς καλύτερα να χρησιμοποιήσουμε Off-chain validation. Σε αντίθετη περίπτωση, ίσως είναι καλύτερα να χρησιμοποιηθεί συνδυασμός των μεθόδων. Δηλαδή ορισμένες αποδείξεις να επικυρώνονται 0ff-chain και κάποιες On-chain, ένα υβριδικό σχήμα που θα μας προσφέρει και ασφάλεια και

μια καλύτερη απόδοση.

Βιβλιογραφία

[1] Andreas M. Antonopoulos and Gavin Wood. *Mastering Ethereum: building smart contracts and DApp*. 2018.

[2] Blockchain Applications. https://blockgeeks.com/guides/blockchain-applications/.

[3] Blockchain. https://www.upgrad.com/blog/why-blockchain-is-important/.

[4] Cloudera Blog. https://blog.cloudera.com/understanding-hdfs-recovery-processes-part-1/.

[5] Ederov Boris. Merkle tree traversal techniques. 04 2007.

[6] Elliptic Curve. https://fangpenlin.com/posts/2019/10/07/elliptic-curve-cryptography-explained/.

[7] Katerina Doka, Tasos Bakogiannis, Ioannis Mytilinis, and Georgios Goumas. *CloudAgora: Democratizing the Cloud*, pages 142–156. 06 2019.

[8] Wikipedia for Digital signatures. https://en.wikipedia.org/wiki/digital_signature.

[9] Geek for Geeks. https://www.geeksforgeeks.org/what-is-dfsdistributed-file-system/.

[10] Wikipedia Crypto Hash Functions. https://en.wikipedia.org/wiki/cryptographic_hash_function.

[11] Apache Hadoop. https://hadoop.apache.org.

[12] Apache Hadoop HDFS. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/hdfsdesign.html.

[13] Investopedia. https://www.investopedia.com/terms/b/blockchain.aspwhat-is-blockchain.

[14] Iveta Kremenova and Milan Gajdos. Decentralized networks: The future internet. *Mobile Networks and Applications*, 24, 01 2019.

[15] Medium. https://medium.com/@zhaohuabing/hash-pointers-and-data-structures-f85d5fe91659.

[16] Ralph Merkle. A digital signature based on a conventional encryption function. *LNCS*, 293:369–378, 08 1987.

[17] Muqaddas Naz, Nadeem Javaid, and Sohail Iqbal. *Research Based Data Rights Management Using Blockchain Over Ethereum Network*. PhD thesis, 09 2019.

[18] Merkle proofs. https://medium.com/crypto-0-nite/merkle-proofs-explained-6dd429623dc5.

[19] John Taylor Rowan, Garnier and. *Discrete Mathematics: Proofs, Structures and Applications*. 2009.

[20] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Rob Chansler. The hadoop distributed file system. *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on: 2010*, 26, 05 2010.

[21] DIstributed Systems. https://www.freecodecamp.org/news/a-thorough-introduction-to-distributed-systems-3b91562c9b3c/.

[22] Michael Szydlo. Merkle tree traversal in log space and time. volume 3027, pages 541–554, 05 2004.

[23] Hawaii University. http://itm-vm.shidler.hawaii.edu/hdfs.

[24] Maarten van Steen and Andrew Tanenbaum. A brief introduction to distributed systems. *Computing*, 08 2016.

[25] T White. *Hadoop: The Definitive Guide*. 01 2010.

[26] Wikipedia. https://en.wikipedia.org/wiki/distributed$_{f}ile_{s}ystem_{(}microsoft)$.

[27] K. Ziegler. A distributed information system study. *IBM Systems Journal*, 18(3):374–401, 1979.

National Technical University of Athens
School of Electrical and Computer Engineering
Department of Information Technology and Computers

Verifiable Distributed Storage Using Blockchain

DIPLOMA PROJECT

ALYZIA-MARIA KONSTA

Supervisor : Nectarios Koziris
NTUA Proffesor

Athens, October 2020

National Technical University of Athens
School of Electrical and Computer Engineering
Department of Information Technology and Computers

Verifiable Distributed Storage Using Blockchain

DIPLOMA PROJECT

ALYZIA-MARIA KONSTA

Supervisor : Nectarios Koziris
NTUA Proffesor

Approved by the examining committee on the October 29, 2020.

Nectarios Koziris NTUA Professor	Giorgos Gkoumas NTUA As.Professor	Dionisios Pnevmatikatos NTUA Professor

Athens, October 2020

Alyzia-Maria Konsta
Electrical and Computer Engineer

Abstract

The purpose of the diploma thesis is to develop a data storage system, which provides high performance while ensuring data security. To achieve high performance but also for scalability in relation to the volume of data, we usually resort to distributed approaches such as HDFS or Google GFS. However, as there is no longer a central authority that we trust but in a distributed system nodes can run from different physical entities, the issue of data trust and security arises. To deal with this problem, in this work we use blockchain technology. The ability to execute code in a verifiable way (smart contracts) has led to the adoption of blockchains in a variety of applications beyond cryptocurrencies. The system we developed integrates the Ethereum blockchain into HDFS and ensures that no node can modify / delete files without doing so through the HDFS protocol. The experimental evaluation of the system showed that security does not come for free but we have to sacrifice part of the performance. More specifically, we pay on time an additional x% when importing files and a y% on system availability.

Key words

Secure File System, Merkle Trees, Blockchain, Distributed File System, HDFS.

Acknowledgements

I would like to thank all the people who supported my work and helped me get results of better quality. I am also grateful to the supervisor of my thesis professor Nectarios Koziris, for the trust he showed me with the assignment of this diploma thesis. I would also like to thank Katerina Doka and Ioannis Mytilinis for the excellent cooperation, their patience and support in overcoming numerous obstacles I have been facing through my research .

Last but not least, I would like to thank my family and friends for their support throughout my studies at the National Technical University of Athens.

Alyzia-Maria Konsta,

Athens, October 29, 2020

Contents

List of Figures

Chapter 1

Introduction

1.1 About this project

Now days the need of storing and processing big amount of data is necessary. The applications being used by people and companies presuppose the processing and possession of multiple information. The amount of this information require the use of a cluster of computers, since it can not be stored or processed by only one computer. This need brought to the limelight the distributing systems, more precisely the distributing file systems for storing such a big amount of data. Distributing file systems function like one local file system without being perceived by the user the distributing nature of the system. As their function has spread beyond technology world a variety of applications are using them. As a result remote computers have in their possession parts of our files.

For the time being most applications online tend to consider the users reliable. The users are being inspected by one master-node, which considers to be the "central authority" of the system. This node, adjusts and tunes the system and considers to be reliable. The gathering of all responsibilities to one node can be fatal for the proper operation of the system, since this node may be malicious. So in the case of distributing file systems our files may be deleted or tampered, without being perceived since retrieving the file. The security of those systems is very important for the users. They should know that their data are safe and ready to be retrieved whenever they want to read or change their file.

A very common distributed file system is HDFS, which works with master-slave logic. That is, there is a central authority node, Namenode, which coordinates the other worker nodes, the Datanodes, and checks if the system is working properly. HDFS is used in a variety of applications, as its software can also run on home computers.However, the absolute trust given to Namenode can not always ensure the proper functioning of the system, as it could collaborate with a Datanode and provide false information about the proper functioning of the system. For example, to hide the modifications that some files accepted. So we are called to strengthen the system with verification mechanisms and to create a system that can detect any attempt to alter the information either by Namenode or by Datanodes.

To achieve data security we use Merkle Trees, which are widely used to validate large data structures and take advantage of Blockchain. Transparency of transactions and the lack of central authority offered by Blockchain technology, has spread its use in a variety of applications. Also the data stored in Blockchain can not be deleted or modified, so it is a reliable source of information that no node of the system can be abused.

1.1.1 System's Usage

Cloud computing technology dominates the IT world globally, allowing organizations and individuals to use remote resources, whether they are for storage, computing or applications purpose instead of local systems which can be costly in performance and resources.

Cloud services are mainly based on a few providers acting as reliable nodes for data transfer, storage and processing. In this way users must show complete trust in the providers for the storage and processing of their data. Providers gather a large amount of information that they can use to their

advantage. Also, large amounts of money are consumed for the implementation of such large systems, so that they are able to serve a large number of customers.

Some of the missing features of these systems can be provided by Blockchain technology. More specifically, strong guarantees of evidence-based data integrity rather than just customer trust to the provider and the operation of a fully decentralized system [7].

HDFS shares files on remote computers. A number of nodes are involved and it works with master-slave logic, as mentioned above. The participating nodes can be any computers that we have no reason to trust. Suppose there is such a system in which we would like to participate to store some files that are impossible due to size to be stored locally on our computer. This system involves local computers, ordinary users and not companies that collect the entire volume of data on specific nodes. So we take the initiative and upload our files to this HDFS Cloud. The file is split into Blocks, which are stored on remote computers. Our file is now under the management of these nodes. Any attempt to modify or change our data by Datanodes (nodes that store files) could go unnoticed. Therefore, when we decide to recover our files we noticed that our files have been modified or even parts of the file have been deleted.

Adding Blockchain to HDFS offers a solution to the problem of reliability and a way to validate data integrity. Smart contracts (programs running on-chain), offer security of operation and accuracy of the results, as they store the results in the Blockchain and everyone can retrieve and control the final result. By using this system, any malicious node will be caught attempting to change the content of our files.

1.1.2 Overview of the System

Namenode receives a BlockReport every 6 hours from each Datanode, the report so far contains a variety of information about the status of Datanode. We have strengthened the report to contain some more information to validate the integrity of the data. The main idea is that every time a Block-Report is activated, Datanode will be notified to provide certain Merkle . Merkle proofs are based on data encryption and are used in large structures to validate their integrity, without the need to own the structure. For each block that Datanode has stored it must send a certain number of proofs, which is requested by Blockchain. Blockchain determines how much and which proofs will be given. BlockReport is then sent to Namenode, which must validate the results. As Namenode receives the BlockReport, it connects to the Blockchain and retrieves what proofs it indicated that the Datanode must give. Because Namenode is unreliable, we implemented an On-chain validation and a local validation. On-chain validation is more secure, as it activates the execution of a smart contract, unlike what is provided by local validation in Namenode. Namenode can corrupt results and provide invalid information to the system. On the other hand, the execution of the smart contract is stored in Blockchain and anyone can read it and confirm the Namenode's action at any time. The algorithm is the same for both cases. If any proof is found to be incorrect, the respective Datanode is obliged to delete the corrupted Block. Clearly on-chain validation offers greater security and reliability, but it costs more over time.

Chapter 2

Distributed Systems

2.1 Overview

The pace at which computer systems are evolving is rapid. From the 40's until late 80's, computers where large and expensive. Furthermore, these computers operated independently from another, due to lack of a way to connect them.

Starting at the 80's, two innovator technologies began to change this situation. The first was the development of powerful multiprocessors .With multi-core CPUs, we now are refacing the challenge of adapting and developing programs to exploit parallelism.

The second development was the innovation of high-speed computer networks. Local-area networks or LANs allow thousands of machines within a building to be connected in such a way that small amounts of information can be transferred in a few microseconds or so. Larger amounts of data can be moved between machines at rates of billions of bits per second(bps).Wide-area networks or WANs allow hundreds of millions of machines all over the earth to be connected at speeds varying from tens of thousands to hundreds of millions bps, and sometimes even faster.

Parallel to the development of increasingly powerful and networked machines, we have also been able to witness miniaturization of computer systems. The result of these technologies is that it is now not only feasible, but easy, to put together a computing system composed of many networked computers. These computers are generally geographically dispersed, for which reason they are usually said to form a distributed system. The size of a distributed system may vary from a handful of devices, to millions of computers. The interconnection network may be wired, wireless, or a combination of both. Moreover, distributed systems are often highly dynamic, in the sense that computers can join and leave, with the topology and performance of the underlying network almost continuously changing.

2.1.1 What is a distributing system

Various definitions of distributed systems have been given in the literature. For our purposes it is sufficient to give a loose characterization:A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system.This definition refers to two characteristic features of distributed systems. The first one is that a distributed system is a collection of computing elements each being able to behave independently of each other. A computing element, which we will generally refer to as a node, can be either a hardware device or a software process. A second element is that users (be they people or applications) believe they are dealing with a single system. This means that one way or another the autonomous nodes need to collaborate. How to establish this collaboration lies at the heart of developing distributed systems. Note that we are not making any assumptions concerning the type of nodes. In principle, even within a single system, they could range from high-performance mainframe computers to small devices in sensor networks. Likewise, we make no assumptions concerning the way that nodes are interconnected.

2.1.2 Goals of Distributed Systems

Sometimes we are forced to build distributed systems because the resources we need are located in different machines.The very nature of an application may require the use of a communication network that connects several computers: for example, data produced in one physical location and required in another location.An important goal of a distributed system is to make it easy for users (and applications)to access and share remote resources. Resources can be virtually anything, but typical examples include peripherals, storage facilities, data, files, services, and networks, to name just a few. There are many reasons for wanting to share resources. One obvious reason is that of economics. For example, it is cheaper to have a single high-end reliable storage facility be shared then having to buy and maintain storage for each user separately.

Connecting users and resources also makes it easier to collaborate and exchange information, as is illustrated by the success of the Internet with its simple protocols for exchanging files, mail, documents, audio, and video. The connectivity of the Internet has allowed geographically widely dispersed groups of people work together by means of all kinds of groupware, that is, software for collaborative editing, teleconferencing,and so on, as is illustrated by multinational software-development companies that have outsourced much of their code production to Asia.However, resource sharing in distributed systems is perhaps best illustrated by the success of file-sharing peer-to-peer networks. These distributed systems make it extremely simple for users to share files across the Internet. Peer-to-peer networks are often associated with distribution of media files such as audio and video. In other cases, the technology is used for distributing large amounts of data,as in the case of software updates, backup services, and data synchronization across multiple servers.

An important goal of a distributed system is to hide the fact that its processes and resources are physically distributed across multiple computers, possibly separated by large distances. In other words, it tries to make the distribution of processes and resources transparent, that is, invisible, to end users and applications. The concept of transparency can be applied to several aspects of a distributed system,of which the most important ones are listed on the below Figure. We use the term object to mean either a process or a resource.

Transparency	Description
Access	Hide differences in data representation and how an object is accessed
Location	Hide where an object is located
Relocation	Hide that an object may be moved to another location while in use
Migration	Hide that an object may move to another location
Replication	Hide that an object is replicated
Concurrency	Hide that an object may be shared by several independent users
Failure	Hide the failure and recovery of an object

Figure 2.1: Transparent Distribution

For many of us worldwide connectivity through the Internet is as common as being able to send a postcard to anyone anywhere around the world. Moreover, where until recently we were used to having relatively powerful desktop computers for office applications and storage, we are now witnessing that such applications and services are being placed in what has been coined “the cloud,” in turn leading to an increase of much smaller networked devices such as tablet computers. With this in mind, scalability has become one of the most important design goals for developers of distributed systems. Scalability of a system can be measured along at least three different dimensions:

Size scalability: A system can be scalable with respect to its size, meaning that we can easily add

more users and resources to the system without any noticeable loss of performance.

Geographical scalability: A geographically scalable system is one in which the users and resources may lie far apart, but the fact that communication delays may be significant is hardly noticed.

Administrative scalability: An administratively scalable system is one that can still be easily managed even if it spans many independent administrative organizations.

2.2 Types of Distributing Systems

Let's take a closer look at the various types of distributed systems. We make a distinction between distributed computing systems, distributed information systems,and pervasive systems (which are naturally distributed).

• Distributed computing is a computing concept that, in its most general sense, refers to multiple computer systems working on a single problem. In distributed computing, a single problem is divided into many parts, and each part is solved by different computers. As long as the computers are networked, they can communicate with each other to solve the problem. If done properly, the computers perform like a single entity. The ultimate goal of distributed computing is to maximize performance by connecting users and IT resources in a cost-effective, transparent and reliable manner. It also ensures fault tolerance and enables resource accessibility in the event that one of the components fails.

• A distributed information system purportedly allows flexibility by reducing implementation constraints of traditional centralized information systems. The intent of such systems is to provide a logically integrated information system while providing for physical distribution of data over two or more computing facilities. Thus, an authorized user can access data from any of the participating computing facilities in the same manner as from a centralized information system.

• Pervasive systems are intended to naturally blend into our environment. They are naturally also distributed systems. What makes them unique in comparison to the computing and information systems described so far, is that the separation between users and system components is much more blurred. There i soften no single dedicated interface, such as a screen/keyboard combination. Instead, a pervasive system is often equipped with many sensors that pick up various aspects of a user's behavior [27, 24].

2.3 Distributed Applications

In the next section, we are going to describe the functionality of some distributed applications.

2.3.1 Map Reduce

Hadoop MapReduce is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.

A MapReduce job usually splits the input data-set into independent chunks which are processed by the map tasks in a completely parallel manner. The framework sorts the outputs of the maps, which are then input to the reduce tasks. Typically both the input and the output of the job are stored in a file-system. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks.

Typically the compute nodes and the storage nodes are the same, that is, the MapReduce framework and the Hadoop Distributed File System (see HDFS Architecture Guide) are running on the same set of nodes. This configuration allows the framework to effectively schedule tasks on the nodes where data is already present, resulting in very high aggregate bandwidth across the cluster.

The MapReduce framework consists of a single master ResourceManager, one worker NodeManager per cluster-node, and MRAppMaster per application (see YARN Architecture Guide).

Minimally, applications specify the input/output locations and supply map and reduce functions via implementations of appropriate interfaces and/or abstract-classes. These, and other job parameters, comprise the job configuration.'

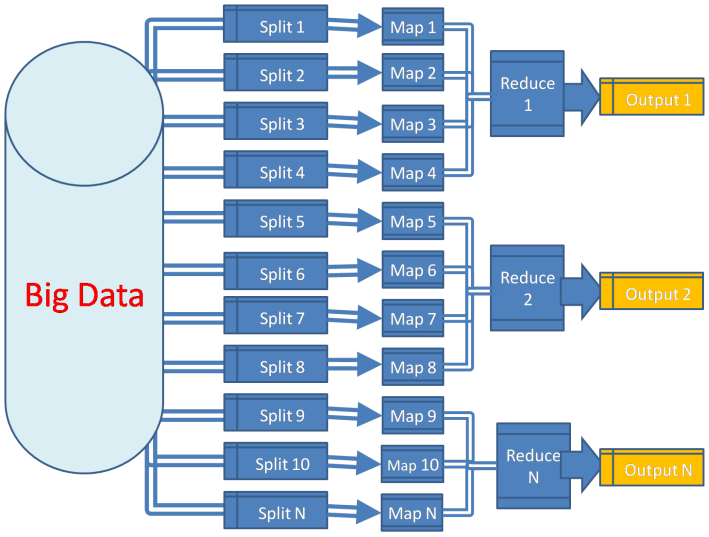


Figure 2.2: MapReduce
http://spideropsnet.com/site1/blog/2014/08/25/apache-hadoop-mapreduce/

2.3.2 Distributed file systems

Distributed file systems can be thought of as distributed data stores. They're storing and accessing a large amount of data across a cluster of machines all appearing as one. They typically go hand in hand with Distributed Computing.

HDFS

Hadoop Distributed File System (HDFS) is the distributed file system used for distributed computing via the Hadoop framework. Boasting widespread adoption, it is used to store and replicate large files (GB or TB in size) across many machines. Its architecture consists mainly of NameNodes and DataNodes. NameNodes are responsible for keeping metadata about the cluster, like which node contains which file blocks. They act as coordinators for the network by figuring out where best to store and replicate files, tracking the system's health. DataNodes simply store files and execute commands like replicating a file, writing a new one and others. We are going to discuss further about HDFS in the next chapters.

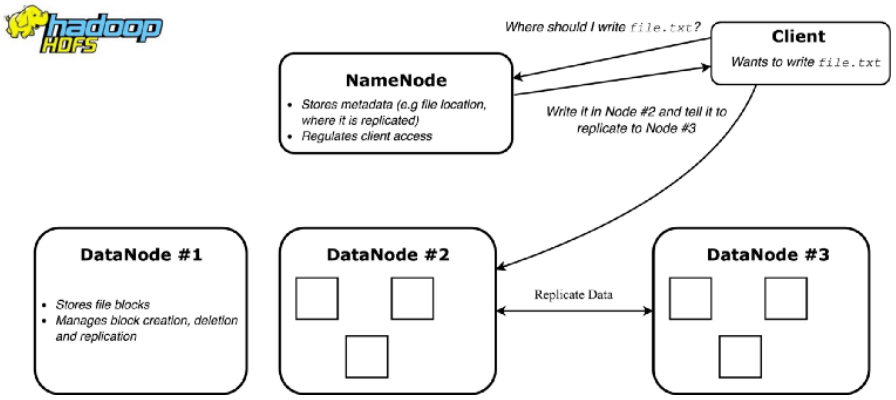


Figure 2.3: HDFS
https://www.freecodecamp.org/news/a-thorough-introduction-to-distributed-systems-3b91562c9b3c/

IPFS

Interplanetary File System (IPFS) is an exciting new peer-to-peer protocol/network for a distributed file system. Leveraging Blockchain technology, it boasts a completely decentralized architecture with no single owner nor point of failure.

IPFS offers a naming system (similar to DNS) called IPNS and lets users easily access information. It stores file via historic versioning, similar to how Git does. This allows for accessing all of a file's previous states[21].

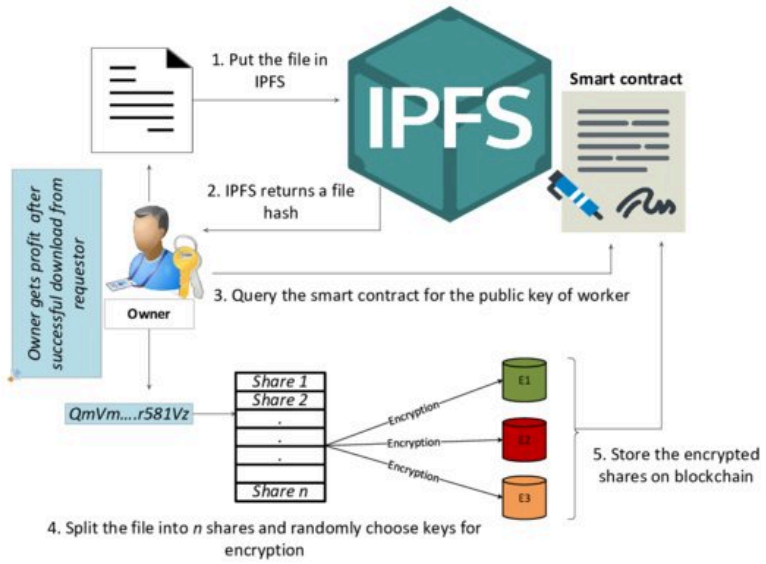


Figure 2.4: IPFS
Naz, Muqaddas, Javaid, Nadeem, Iqbal, Sohail. (2019). Research Based Data Rights Management Using Blockchain Over Ethereum Network.

2.4 Importance of DFS

A file system is a process responsible of the storage and the organization of files being used from a computer. Technically, it organizes these files in a database for the purpose of their storage, organi-

zation, management and the retrieval from the operating system of the computer. File systems can be local or distributed. A Distributed File System, is a file system that is distributed on multiple file servers. It allows users to access or store files, like the local one, from any network. DFS in general is a distributed implementation of the classical file system. A DFS should include the following features:

• Location transparency: There is no need for the client to know about the number or locations of the storage devices. There is a cluster of storage devices in order to provide the system with availability. Also, If a file is copied on multiple nodes, both the copies of the file and their locations should be hidden from one node to another.

• Access Transparency: The files should be accessible anytime like the local ones, without the client realizing that the file is distributed on remote computers. The file system should be automatically located on the accessed file and send it to the client's side.

• Performance: As we mentioned above the client must not realize the difference between a local and a distributed file system. This fact also includes the performance of a DFS. Performance is based on the average amount of time needed to execute the clients request. This time covers the CPU time + time taken to access secondary storage + network access time. It is obvious that a DFS may be slower than a centralized system but it is advisable that the performance of the Distributed File System be similar to that of a local one.

• High Availability: A distributed file system should be able to continue working in case of a partial failure. Hence it is recommended the files be replicated in different nodes in the system. In case of a node crush, the file will be retrieved from a different node. A high authentic and adaptable distributed file system should have different and independent file servers for controlling different and independent storage devices.

Local file systems provides the data quickly, but does not have the ability of storing a huge amount of data. The need of storing very large files have grown over the past years, so the use of distributed file systems is necessary. The DFS is an important tool in order to handle huge amount of data. Also allows multiple users to access or store data remotely, which is very important for big projects and people working in different locations [9].

2.4.1 Applications of DFS

The server component of the Distributed File System was initially introduced as an add-on feature. It was added to Windows NT 4.0 Server and was known as "DFS 4.1". Then later on it was included as a standard component for all editions of Windows 2000 Server. Client-side support has been included in Windows NT 4.0 and also in later on version of Windows. Linux kernels 2.6.14 and versions after it come with an SMB client VFS known as "cifs" which supports DFS. Mac OS X 10.7 (lion) and onwards supports Mac OS X DFS. Many applications have been developed since then such as:

• NFS: Network File System (NFS) is a distributed file system protocol originally developed by Sun Microsystems (Sun) in 1984, allowing a user on a client computer to access files over a computer network much like local storage is accessed. NFS, like many other protocols, builds on the Open Network Computing Remote Procedure Call (ONC RPC) system. The NFS is an open standard defined in a Request for Comments (RFC), allowing anyone to implement the protocol.

• CIFS stands for Common Internet File System. CIFS is an accent of SMB. That is, CIFS is an application of SIMB protocol, designed by Microsoft.

• SMB: SMB stands for Server Message Block. It is a protocol for sharing a file and was invented by IMB. The SMB protocol was created to allow computers to perform read and write operations on files to a remote host over a Local Area Network (LAN). The directories present in the remote host can be accessed via SMB and are called as "shares".

• Hadoop: Hadoop is a group of open-source software services. It gives a software framework for distributed storage and operating of big data using the MapReduce programming model. The core of Hadoop contains a storage part, known as Hadoop Distributed File System (HDFS), and an operating part which is a MapReduce programming model.

• NetWare: NetWare is an abandon computer network operating system developed by Novell, Inc. It primarily used combined multitasking to run different services on a personal computer, using the IPX network protocol [9, 26].

2.5 HDFS

2.5.1 HDFS Overview

Hadoop is an open-source software framework for storing data and running applications on clusters of commodity hardware. It provides massive storage for any kind of data, enormous processing power and the ability to handle virtually limitless concurrent tasks or jobs.

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets. HDFS relaxes a few POSIX requirements to enable streaming access to file system data. HDFS was originally built as infrastructure for the Apache Nutch web search engine project. HDFS is part of the Apache Hadoop Core project [11].

2.5.2 HDFS Architecture

Namenode

The HDFS namespace is a hierarchy of files and directories. Files and directories are represented on the NameNode by inodes, which record attributes like permissions, modification and access times, namespace and disk space quotas. The file content is split into large blocks (typically 128 megabytes, but user selectable file-by-file) and each block of the file is independently replicated at multiple DataNodes (typically three, but user selectable file-by-file). The NameNode maintains the namespace tree and the mapping of file blocks to DataNodes (the physical location of file data). An HDFS client wanting to read a file first contacts the NameNode for the locations of data blocks comprising the file and then reads block contents from the DataNode closest to the client. When writing data, the client requests the NameNode to nominate a suite of three DataNodes to host the block replicas. The client then writes data to the DataNodes in a pipeline fashion. The current design has a single NameNode for each cluster. The cluster can have thousands of DataNodes and tens of thousands of HDFS clients per cluster, as each DataNode may execute multiple application tasks concurrently.

HDFS keeps the entire namespace in RAM. The inode data and the list of blocks belonging to each file comprise the metadata of the name system called the image. The persistent record of the image stored in the local host's native files system is called a checkpoint. The NameNode also stores the modification log of the image called the journal in the local host's native file system. For improved durability, redundant copies of the checkpoint and journal can be made at other servers. During restarts the NameNode restores the namespace by reading the namespace and replaying the journal. The locations of block replicas may change over time and are not part of the persistent checkpoint [20].

Datanode

Each block replica on a DataNode is represented by two files in the local host's native file system. The first file contains the data itself and the second file is block's metadata including checksums for the block data and the block's generation stamp. The size of the data file equals the actual length of the block and does not require extra space to round it up to the nominal block size as in traditional file systems. Thus, if a block is half full it needs only half of the space of the full block on the local drive.

During startup each DataNode connects to the NameNode and performs a handshake. The purpose of the handshake is to verify the namespace ID and the software version of the DataNode. If either

does not match that of the NameNode the DataNode automatically shuts down. The namespace ID is assigned to the file system instance when it is formatted. The namespace ID is persistently stored on all nodes of the cluster. Nodes with a different namespace ID will not be able to join the cluster, thus preserving the integrity of the file system. The consistency of software versions is important because incompatible version may cause data corruption or loss, and on large clusters of thousands of machines it is easy to overlook nodes that did not shut down properly prior to the software upgrade or were not available during the upgrade. A DataNode that is newly initialized and without any namespace ID is permitted to join the cluster and receive the cluster's namespace ID.

After the handshake the DataNode registers with the NameNode. DataNodes persistently store their unique storage IDs. The storage ID is an internal identifier of the DataNode, which makes it recognizable even if it is restarted with a different IP address or port. The storage ID is assigned to the DataNode when it registers with the NameNode for the first time and never changes after that. A DataNode identifies block replicas in its possession to the NameNode by sending a block report. A block report contains the block id, the generation stamp and the length for each block replica the server hosts. The first block report is sent immediately after the DataNode registration. Subsequent block reports are sent every hour and provide the NameNode with an up-todate view of where block replicas are located on the cluster. During normal operation DataNodes send heartbeats to the NameNode to confirm that the DataNode is operating and the block replicas it hosts are available. The default heartbeat interval is three seconds. If the NameNode does not receive a heartbeat from a DataNode in ten minutes the NameNode considers the DataNode to be out of service and the block replicas hosted by that DataNode to be unavailable. The NameNode then schedules creation of new replicas of those blocks on other DataNodes. Heartbeats from a DataNode also carry information about total storage capacity, fraction of storage in use, and the number of data transfers currently in progress. These statistics are used for the NameNode's space allocation and load balancing decisions.

The NameNode does not directly call DataNodes. It uses replies to heartbeats to send instructions to the DataNodes. The instructions include commands to:

- replicate blocks to other nodes;
- remove local block replicas;
- re-register or to shut down the node;
- send an immediate block report.

These commands are important for maintaining the overall system integrity and therefore it is critical to keep heartbeats frequent even on big clusters. The NameNode can process thousands of heartbeats per second without affecting other NameNode operations [20].

Client

User applications access the file system using the HDFS client, a code library that exports the HDFS file system interface. Similar to most conventional file systems, HDFS supports operations to read, write and delete files, and operations to create and delete directories. The user references files and directories by paths in the namespace. The user application generally does not need to know that file system metadata and storage are on different servers, or that blocks have multiple replicas.

When an application reads a file, the HDFS client first asks the NameNode for the list of DataNodes that host replicas of the blocks of the file. It then contacts a DataNode directly and requests the transfer of the desired block.

When a client writes, it first asks the NameNode to choose DataNodes to host replicas of the first block of the file. The client organizes a pipeline from node-to-node and sends the data. When the first block is filled, the client requests new DataNodes to be chosen to host replicas of the next block. A new pipeline is organized, and the client sends the further bytes of the file. Each choice of DataNodes is likely to be different. The interactions among the client, the NameNode and the DataNodes are illustrated in Fig. 1.

Unlike conventional file systems, HDFS provides an API that exposes the locations of a file blocks. It also allows an application to set the replication factor of a file. By default a file's replication factor

is three. For critical files or files which are accessed very often, having a higher replication factor improves their tolerance against faults and increase their read bandwidth [12].

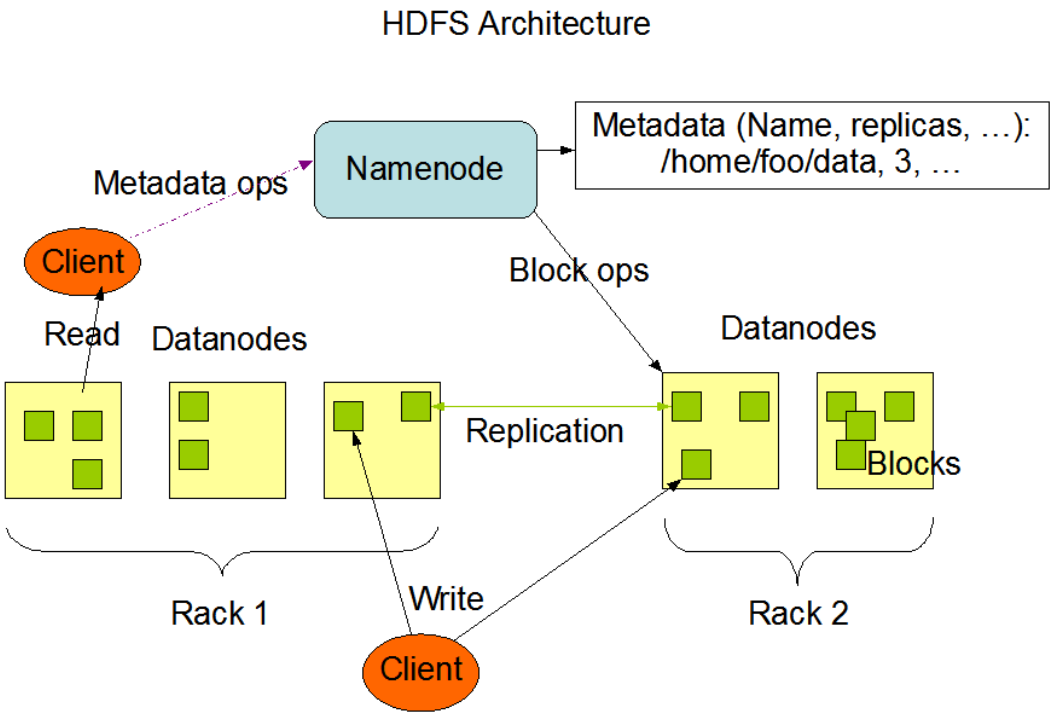


Figure 2.5: HDFS architecture the basic concept

Image and Journal

The namespace image is the file system metadata that describes the organization of application data as directories and files. A persistent record of the image written to disk is called a checkpoint. The journal is a write-ahead commit log for changes to the file system that must be persistent. For each client-initiated transaction, the change is recorded in the journal, and the journal file is flushed and synched before the change is committed to the HDFS client. The checkpoint file is never changed by the NameNode; it is replaced in its entirety when a new checkpoint is created during restart, when requested by the administrator, or by the CheckpointNode described in the next subsection. During startup the NameNode initializes the namespace image from the checkpoint, and then replays changes from the journal until the image is up-to-date with the last state of the file system. A new checkpoint and empty journal are written back to the storage directories before the NameNode starts serving clients.

If either the checkpoint or the journal is missing, or becomes corrupt, the namespace information will be lost partly or entirely. In order to preserve this critical information HDFS can be configured to store the checkpoint and journal in multiple storage directories. Recommended practice is to place the directories on different volumes, and for one storage directory to be on a remote NFS server. The first choice prevents loss from single volume failures, and the second choice protects against failure of the entire node. If the NameNode encounters an error writing the journal to one of the storage directories it automatically excludes that directory from the list of storage directories. The NameNode automatically shuts itself down if no storage directory is available.

The NameNode is a multithreaded system and processes requests simultaneously from multiple clients. Saving a transaction to disk becomes a bottleneck since all other threads need to wait until the synchronous flush-and-sync procedure initiated by one of them is complete. In order to optimize

this process the NameNode batches multiple transactions initiated by different clients. When one of the NameNode's threads initiates a flush-and-sync operation, all transactions batched at that time are committed together. Remaining threads only need to check that their transactions have been saved and do not need to initiate a flush-and-sync operation [20].

Secondary Namenode

Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large. The secondary namenode usually runs on a separate physical machine because it requires plenty of CPU and as much memory as the namenode to perform the merge. It keeps a copy of the merged name□ space image, which can be used in the event of the namenode failing. However, the state of the secondary namenode lags that of the primary, so in the event of total failure of the primary, data loss is almost certain. The usual course of action in this case is to copy the namenode's metadata files that are on NFS to the secondary and run it as the new primary [25].

Block

Filesystems for a single disk build on this by dealing with data in blocks, which are an integral multiple of the disk block size. Filesystem blocks are typically a few kilobytes in size, whereas disk blocks are normally 512 bytes. This is generally transparent to the filesystem user who is simply reading or writing a file of whatever length. However, there are tools to perform filesystem maintenance, such as df and fsck, that operate on the filesystem block level.

HDFS, too, has the concept of a block, but it is a much larger unit—128 MB by default. Like in a filesystem for a single disk, files in HDFS are broken into block-sized chunks, which are stored as independent units. Unlike a filesystem for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block's worth of under□ lying storage. (For example, a 1 MB file stored with a block size of 128 MB uses 1 MB of disk space, not 128 MB.)

Every block has an associated generation stamp, A GS is a monotonically increasing 8-byte number for each block that is maintained persistently by the NameNode. The GS for a block and replica is introduced for the following purposes:

●Detecting stale replica(s) of a block: that is, when the replica GS is older than the block GS, which might happen when, for example, an append operation is somehow skipped at the replica.

●Detecting outdated replica(s) on DataNodes which have been dead for long time and rejoin the cluster.

The blocks are stored in the datanodes a replica in the DataNode context can be in one of the following states:

• FINALIZED: when a replica is in this state, writing to the replica is finished and the data in the replica is "frozen" (the length is finalized), unless the replica is re-opened for append. All finalized replicas of a block with the same generation stamp (referred to as the GS and defined below) should have the same data. The GS of a finalized replica may be incremented as a result of recovery.

• RBW (Replica Being Written): this is the state of any replica that is being written, whether the file was created for write or re-opened for append. An RBW replica is always the last block of an open file. The data is still being written to the replica and it is not yet finalized. The data (not necessarily all of it) of an RBW replica is visible to reader clients. If any failures occur, an attempt will be made to preserve the data in an RBW replica.

• RWR (Replica Waiting to be Recovered): If a DataNode dies and restarts, all its RBW replicas will be changed to the RWR state. An RWR replica will either become outdated and therefore discarded, or will participate in lease recovery.

• RUR (Replica Under Recovery): A non-TEMPORARY replica will be changed to the RUR state when it is participating in lease recovery.

• TEMPORARY: a temporary replica is created for the purpose of block replication (either by replication monitor or cluster balancer). It's similar to an RBW replica, except that its data is invisible to all reader clients. If the block replication fails, a TEMPORARY replica will be deleted.

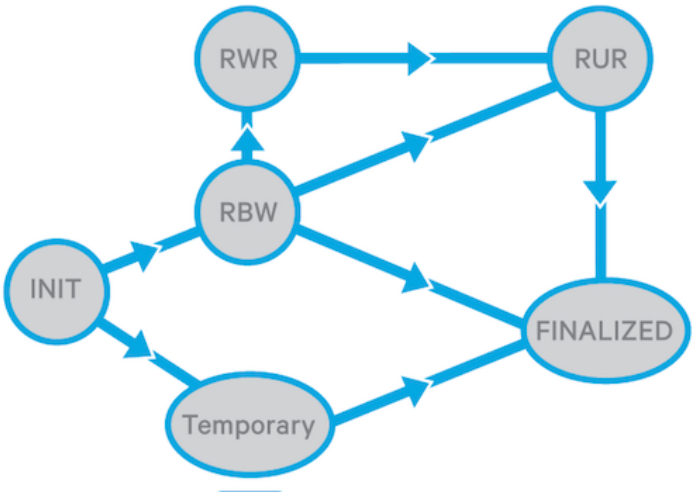


Figure 2.6: Simplified Replica State Trantition

A block in the NameNode context may be in one the following states:

•UNDER CONSTRUCTION: this is the state when it is being written to. An UNDER CONSTRUCTION block is the last block of an open file; its length and generation stamp are still mutable, and its data (not necessarily all of it) is visible to readers. An UNDER CONSTRUCTION block in the NameNode keeps track of the write pipeline (the locations of valid RBW replicas), and the locations of its RWR replicas.

•UNDER RECOVERY: If the last block of a file is in UNDER CONSTRUCTION state when the corresponding client's lease expires, then it will be changed to UNDER RECOVERY state when block recovery starts.

•COMMITTED: COMMITTED means that a block's data and generation stamp are no longer mutable (unless it is reopened for append), and there are fewer than the minimal-replication number of DataNodes that have reported FINALIZED replicas of same GS/length. In order to service read requests, a COMMITTED block must keep track of the locations of RBW replicas, the GS and the length of its FINALIZED replicas. An UNDER CONSTRUCTION block is changed to COMMITTED when the NameNode is asked by the client to add a new block to the file, or to close the file. If the last or the second-to-last blocks are in COMMITTED state, the file cannot be closed and the client has to retry.

•COMPLETE: A COMMITTED block changes to COMPLETE when the NameNode has seen the minimum replication number of FINALIZED replicas of matching GS/length. A file can be closed only when all its blocks become COMPLETE. A block may be forced to the COMPLETE state even if it doesn't have the minimal replication number of replicas, for example, when a client asks for a new block, and the previous block is not yet COMPLETE [4].

2.5.3 HDFS Communication

Application code AND Client

HDFS provides a Java API for applications to use. Fundamentally, the application uses the standard java.io interface. A C language wrapper for this Java API is also available.

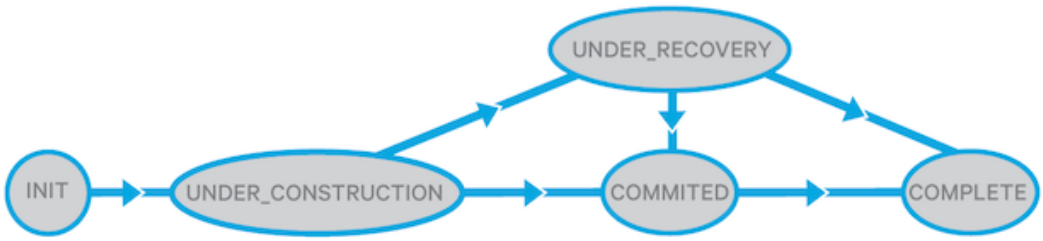


Figure 2.7: Simplified Block State Trantition

Client AND Namenode

The connection between NameNode and the client is governed by the Client Protocol documented in \hdfs\protocol\ClientProtocol.java. A client establishes a connection to a configurable TCP port on the NameNode machine. This is an RPC connection. The communication between the Client and the NameNode uses the ClientProtocol.

The major functions of the Client Protocol are:

• Create : The client asks NameNode to create a new file entry in the namespace. NameNode throws an exception if it is unable to create the file entry. When a file is created, the client is granted a lease on the file. The lease prevents other threads from writing to the file but not from reading from it. A lease has an expiration time (the default value is one minute) and it is the responsibility of the client to renew the lease before it expires. The client sets an interrupt at half the lease life so that it can ask for a renewal if necessary. The lease can be renewed explicitly through the ClientProtocol or implicitly through AddBlock.

• Append : differs from simply writing to the end of a file in two ways: how concurrency is treated. When performing a write, only a single writer is allowed. When performing an append, multiple simultaneous appends are allowed. Where the new block is written. When performing a write, the client specifies the offset of the new block. When performing an append, HDFS chooses the location for the block. These two differences are related because if multiple simultaneous appends occur, the contents of one replica may differ in order from the contents of another replica. If the appends are successful, then all of the contents are contained in each replica, although potentially in a different order.

• AddBlock : requests the NameNode to allocate a new block and return the list of DataNodes the block data should be replicated to. The transferring of the data to the DataNodes is described in the Write subsection. AddBlock also commits the previous block by reporting to the NameNode the actual generation (time) stamp and the length of the block that the client has transmitted to Data-Nodes. A write in not considered completed until it has been committed.

• Write : An application request to create a file does not reach the NameNode immediately. In fact, initially the HDFS client caches the file data into a temporary local file. Application writes are transparently redirected to this temporary local file. When the local file accumulates data worth at least one HDFS block size, the client contacts the NameNode to create a file. The NameNode then proceeds as described in the subsection on Create. The client flushes the block of data from the local temporary file to the specified DataNodes. When a file is closed, the remaining un-flushed data in the temporary local file is transferred to the DataNode. The client then tells the NameNode that the file is closed. At this point, the NameNode commits the file creation operation into a persistent store. If the NameNode dies before the file is closed, the file is lost.

The bulk of the work is done in DFSOutputStream.java. It uses the ClientProtocol method addBlock() to allocate blocks for the new data. A set of DataNode locations are returned to the client. Each DataNode is intended to contain one replica of the block.

The client application writes data that is cached internally by this stream. Data is broken up into

packets, each packet is typically 64K in size. A packet is comprised of chunks. Each chunk is typically 512 bytes and has an associated checksum with it.

When a client application fills up the currentPacket, it is enqueued into dataQueue. The DataStreamer thread picks up packets from the dataQueue, sends it to the first DataNode in the pipeline and moves it from the dataQueue to the ackQueue. The ResponseProcessor receives acks from the DataNodes. When a successful ack for a packet is received from all DataNodes, the ResponseProcessor removes the corresponding packet from the ackQueue.

In case of error, all outstanding packets are moved from ackQueue. A new pipeline is set up by eliminating the bad DataNode from the original pipeline. The DataStreamer now starts sending packets from the dataQueue.

• Complete : When the client is done writing data to the given filename, the Complete function is called to close the file. The function complete() in the ClientProtocol returns a Boolean, indicating whether the file has been closed successfully or not. If the function returns false, the caller should try again. The function complete() also commits the last block of the file by reporting to the NameNode.java the actual generation stamp and the length of the block that the client has transmitted to data nodes. A call to complete() will not return true until all the file's blocks have been replicated the minimum number of times. Thus, DataNode failures may cause a client to call complete() several times before succeeding. Finally the lease on the file is terminated.

• Read : From the client's perspective, a read occurs in two steps. First, the ClientProtocol is used to locate the block using LocatedBlock. The NameNode returns a list of locations of the replicates. The client then contacts the DataNodes which hold the replicates to retrieve the data. DataNodes maintain an open server socket so that clients can read or write efficiently. The actual reading of the data is managed by DFSInputStream.java. If one of the DataNodes has failed—i.e. the read from the DataNode times out—the client is responsible for retrieving the desired block from the next DataNode in the replication sequence. It is possible that some of the blocks retrieved from the DataNode are corrupted. In such a case, the client should inform the NameNode of the corrupted block(s) using reportBadBlocks in the Client Protocol. A corrupted block is determined by the use of checksums within the block (as described in the subsection on Write).

Client AND Datanade

A client communicates with a DataNode directly to transfer (send/receive) data using the DataTransferProtocol, defined in DataTransferProtocol.java. For performance purposes this protocol is a streaming protocol, not RPC. The client buffers data until a full block (the default is 64 Mbytes) has been created and then the block is streamed to the DataNode. The DataTransferProtocol defines operations to read a block (opReadBlock()), write a block (opWriteBlock()), replace a block (opReplaceBlock()), copy a block (opCopyBlock()), and to get a block's Checksum (opBlockChecksum()).

Datanode AND Namenode

All communication between Namenode and Datanode is initiated by the Datanode, and responded to by the Namenode. The Namenode never initiates communication to the Datanode, although Namenode responses may include commands to the Datanode that cause it to send further communications. DataNode sends information to NameNode through four major interfaces defined in the DataNodeProtocol. These four are:

• DataNode Registration. The DataNode informs NameNode of its existence. NameNode returns its registration id. This registration id is a parameter of other DataNode functions. Registration is triggered when a new DataNode is initiated, an old one is re-initiated, or when a new NameNode is initiated.

• DataNode sends heartbeat. The DataNode sends a heartbeat message every few seconds. This includes some information statistics about capacity and current activity. NameNode returns a list of block oriented commands for DataNode to execute. These commands primarily consist of instructions

to transfer blocks to other DataNodes for replication purposes, or instructions to delete blocks. The NameNode can also command an immediate Block Report from the DataNode, but this is only done to recover from severe problems.

• DataNode sends block report. DataNode periodically reports the blocks contained in its storage. The period is typically configured to hourly.

• DataNode notifies BlockReceived. DataNode reports that it has received a new block, either from a Client (during file write) or from another DataNode (during replication). It reports each block immediately upon receipt.

2.5.4 Use Cases

In this subsection we provide sequential descriptions of the most important HDFS functions: write and read.

Create

Client:

The client asks NameNode to create a new file entry in the namespace. NameNode throws an exception if it is unable to create the file entry. When a file is created, a lease is also created by NameNode for the file. The client sets an interrupt at half the lease life so that it can ask for a renewal if necessary.

NameNode:

The create() method in the ClientProtocol creates an empty file. It is handed a full path name and, after checking for duplicates, permissions, whether NameNode is in Safe Mode, and a few other things, the INode data structure is created. Subsequently, the full path name is used by the client, via the various ClientProtocol methods, to access this data structure. Once created, the file is available for reading. Deletion, rename, or re-create can only be done after the file has been completed (closed).

The file creation event is added to the EditLog. The log is used in the recovery procedure in case the NameNode fails. See NameNode Backup Mode where the recovery procedure for NameNode is described.

The client is granted a lease as long as there is not another lease holder for this file. A lease ensures that there is only one writer to a file at a time. If the client is not actively using the lease, it will time out after one minute (default value). Management of leases is synchronized to prevent race conditions over leases. See NameNode Lease Management where this is described.

Write

An overview of the client actions as shown in Figure 2.8 is that an application request to write a file does not reach the NameNode immediately. Initially the DFSOutputStream caches the file data into a temporary local file. Application writes are transparently redirected to this temporary local file. When the local file accumulates data worth at least one HDFS block size, DFSOutputStream contacts the NameNode using the protocol method addBlock to add a block to the file and return a pipeline of DataNodes for the client. The client writes the block to the first DataNode in the pipeline which writes the block and on completion of its writing, passes the block to the next DataNode in the pipeline, sends a blockReceived message to the NameNode and an acknowledgement to DFSOutputStream. Subsequent DataNodes in the pipeline repeat the same sequence.

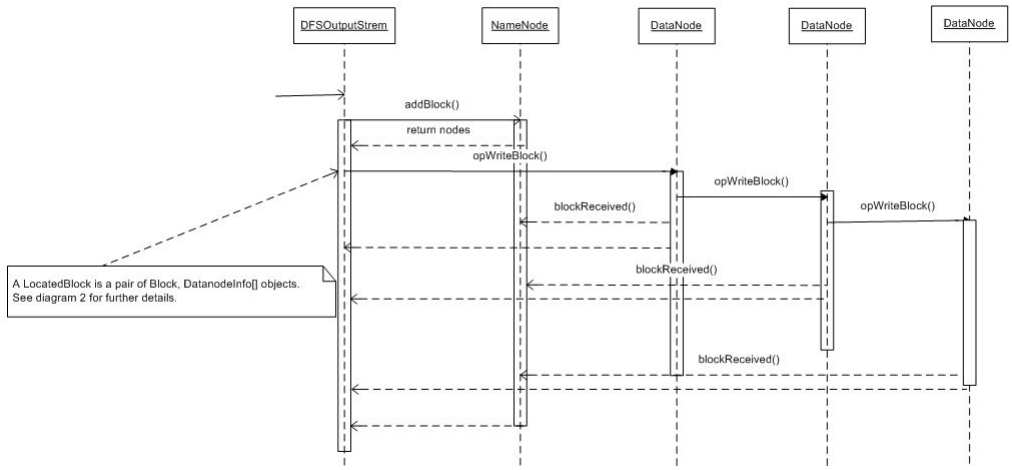


Figure 2.8: Write Overview

In Figure 2.8 we show a drill down of Figure 2.9, detailing the classes used in the sequence. The leftmost red box encloses those objects in the client name space and the rightmost red box encloses those objects in the DataNode name space. Also, for space reasons we do not show the replicated data nodes. Each of them repeats the sequence enclosed in the right red box.

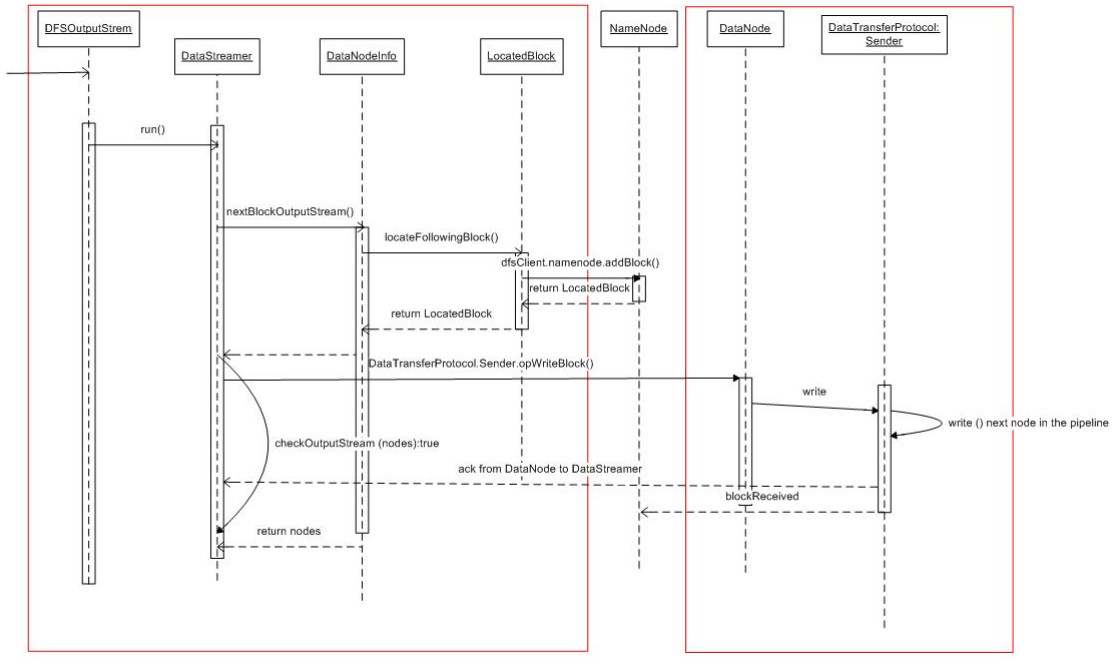


Figure 2.9: Write Detailed Overview

Figure 2.9 shows these operations in more detail. The client application writes data that is cached internally by DFSOutputStream. DFSOutputStream breaks the data into packets, each packet is typically 64K in size. A packet comprises of chunks. Each chunk is typically 512 bytes and has an associated checksum with it. DFSOutputStream creates the thread DataStreamer that is responsible first for communicating with the NameNode to get the pipelines for the DataNodes and secondly for actually transferring the Data to the DataNodes. DataStreamer communicates with NameNode through the nextBlockOutputStream method of the protocol class DataNodeInfo. It in turn calls the

method locateFollowingBlock in the LocatedBlock. The addBlock method in the NameNode protocol is then called to transfer control to the NameNode.

Once NameNode receives a request to add a block, it:

- Verifies the client is the lease holder for this file.
- Cleans up the prior write. This involves

  Committing the prior write

  Renewing the lease

  Modifying EditLog to reflect the new block that has been written
- Allocates replicate number of blocks on DataNodes and stores the DataNode ID and the block number using
- BlockManager (where replicate is a user-settable value for the number of replicas of the file). Returns the allocated block information back to locatedBlock.

DataStreamer then transfers the block to the first DataNode in the pipeline a packet at time. Each packet is enqueued into dataQueue. The DataStreamer thread picks up packets from the dataQueue, sends it to the first DataNode using the DataTransferProtocolSender operation WriteBlock. The DataStreamer also moves packet from the dataQueue to the ackQueue. The ResponseProcessor thread is created by the DataStreamer and it receives acks from the DataNodes When a successful ack for a packet is received from all DataNodes, the ResponseProcessor removes the corresponding packet from the ackQueue.

DataNode performs the following operations on the block:

- Copies the block to its local file system.
- Sends a blockReceived notification to NameNode that it has written a block
- Sends the block to the next DataNode in the pipeline
- Sends an acknowledgement to the DataStreamer that the block has been written.

The next DataNode in the pipeline performs exactly the same operations. This continues until the pipeline is exhausted, i.e. when replicate replicas of the block have been written. The DataStreamer, when it has received acknowledgements from all of the DataNodes, informs the NameNode of a successful write. It does this either through the next call to nextBlockOutputStream or through a close of the file (not shown in Figure 5). This is not all done synchronously with the write request. In particular, the writing of the replicates is done asynchronously.

Read

Client:

From the client's perspective, a read occurs in two steps. First, the ClientProtocol is used to locate the block using LocatedBlock. The NameNode returns a list of locations of the replicates. The client then contacts the DataNodes which hold the replicates to retrieve the data. DataNodes maintain an open server socket so that clients can read or write efficiently. The actual reading of the data is managed by DFSInputStream.java. If one of the DataNodes has failed—i.e. the read from the DataNode times out—the client is responsible for retrieving the desired block from the next DataNode in the replication sequence. It is possible that some of the blocks retrieved from the DataNode are corrupted. In such a case, the client should inform the NameNode of the corrupted block(s) using reportBadBlocks in the Client Protocol. A corrupted block is determined by the use of checksums within the block (as described in the subsection on Write).

DataNode:

The DataNode maintains just one critical table: Block replica -> stream of bytes (of BLOCK SIZE or less) The DataNode Daemon maintains an open server socket so that client code read data [23].

Complete

Client:

When the client is done writing data to the given filename, the Complete function is called to close the file. The function complete() in the ClientProtocol returns a Boolean, indicating whether the file has been closed successfully or not. If the function returns false, the caller should try again. The function complete() also commits the last block of the file by reporting to the NameNode.java the actual generation stamp and the length of the block that the client has transmitted to data nodes.

A call to complete() will not return true until all the file's blocks have been replicated the minimum number of times. Thus, DataNode failures may cause a client to call complete() several times before succeeding.

NameNode:

If all of the blocks have been committed to the log, then the NameNode closes the file, releases the lock, and returns true.

If all of the blocks have not been committed to the log, then the invocation of complete in the ClientProtocol returns false.

Chapter 3

Merkle Trees

3.1 Merkle Trees

3.1.1 Merkle Trees Overview

A Merkle tree is a tree in which every leaf node is labelled with the cryptographic hash of a data block, and every non-leaf node is labelled with the cryptographic hash in the labels of its child nodes. Hash trees allow efficient and secure verification of the contents of large data structures.

Cryptographic hash function

A cryptographic hash function is a hash function that is suitable for use in cryptography. It is a mathematical algorithm that maps data of arbitrary size (often called the ”message”) to a bit string of a fixed size and is a one-way function. Intuitively, a one way function F is one which is easy to compute but difficult to invert. If y = F(x), then given x and F, it is easy to compute y, but given y and F it is effectively impossible to compute x. Properties of a hash function:

- Deterministic : a given input message always produces the same output hash.
- Verifiability : Computing the hash of a message is efficient (linear complexity).
- Noncorrelation : A small (even 1-bit) change to the message should change the hash output so extensivly that it cannot be correlated to the hash of the original message.
- Irreversibility : computing the message of its hash is infeasible.
- Collision Protection : It should be infeasible to calculate two different messages that produce the same hash output [10, 1].

Binary Tree

A binary Tree comprises a triple of sets (L,S,R) where L and R are binary trees (or are empty) and S is a singleton set. The single element of S is the root, and L and R are called respectively, the left and right subtrees of the root. The definition is recursive because it defines a binary tree in terms of the components L,S and R, two of which are themselves binary trees. Thus, L and R, if non-empty, are both defined as triples of the form (L’, S’, R’) and so on. This way of defining binary trees is extremely useful [19].

A complete binary tree T is said to have height H if it has 2^H leaves, and $2^H - 1$ interior nodes. By labeling each left child node with a “0” and each right child node with a “1”, the digits along the path from the root identify each node. Interpreting the string as a binary number, the leaves are naturally indexed by the integers in the range 0, 1,... $2^H - 1$. The higher the leaf index, the further to the right that leaf is. Leaves are said to have height 0, while the height of an interior node is the length of the path to a leaf below it. Thus, the root has height H, and below each node at height h, there are 2h leaves.

Bulding a Merkle Tree

Merkle Proposed a bulding scheme based on a binary tree in [16]. A Merkle tree is a complete binary tree equipped with a function hash and an assignment, Φ , which maps the set of nodes to the set of k-length strings: n $\rightarrow \Phi$(n) $\in \{0,1\}^k$. For the two child nodes, nlef t and nright, of any interior node, nparent, the assignment Φ is required to satisfy

$$\Phi(\text{nparent}) = \text{hash}(\Phi(\text{nlef t}) \,||\, \Phi(\text{nright}))$$

The function hash is a candidate one-way function. For each leaf l, the value Φ(l) may be chosen arbitrarily, and then equation above determines the values of all the interior nodes [22].

Authentication paths: Every leaf has height 0 and the hash function of two leaves has height 1, etc. In this manner the root has height H if the tree has 2^H = N leaves. Then for every leaf the authentication path is the set $\{\text{Auth}_h \mid 0 <= \text{h} < \text{H}\}$. So a leaf can be authenticated as follows: First we apply our hash function f to the leaf and its sibling Auth_0 , then we apply f to the result and Auth_1 , and so on all the way up to the root. If the calculated root value is the same as the published root value, then the leaf value is accepted as authentic. This operation requires $\log_2$(N) invocations of the hash function f [5].

Merkle Proofs

Merkle proofs are used to decide upon the following factors:

- If the data belongs in the merkle tree
- To concisely prove the validity of data being part of a dataset without storing the whole data set
- To ensure the validity of a certain data set being inclusive in a larger data set without revealing either the complete data set or its subset.

Merkle trees make extensive use of one way hashing. Merkle proofs are established by hashing a hash's corresponding hash together and climbing up the tree until you obtain the root hash which is or can be publicly known. Given that one way hashes are intended to be collision free and deterministic algorithms, no two plaintext hashes can/should be the same. Merkle proofs are better explained with the following example. In order to verify the inclusion of data [K], in the merkle tree root, we use a one way function to hash [K] to obtain H(K). In order to validate the inclusivity of K, K doesn't have to be revealed, similarly the hash of data L can be revealed without any implicit security repercussions and so on.

H(K) when hashed with the hash of the unknown dataset L, yields H(KL). H(KL) hashed with H(IJ) leads to H(IJKL). H(IJKL) hashed with H(MNOP) leads to H(IJKLMNOP) H(IJKLMNOP) when hashed with H(ABCDEFGH) yields H(ABCDEFHGIJKLMNOP) which happens to be our publically available merkle root. We've hence proven that the data set K is indeed present in our merkle tree by making use of H(L), H(IJ), H(MNOP) and H(ABCDEFGH) without having to reveal K or any of the data. In order to obtain a merkle proof of H, we need H(L), H(IJ), H(MNOP) and H(ABCDEFGH) with which we can together obtain H(ABCDEFHGIJKLMNOP) hence proving that H(K) was part of the merkle tree implying that data set K was indeed part of the universal dataset [A, B, C, … , N, O, P] [18].

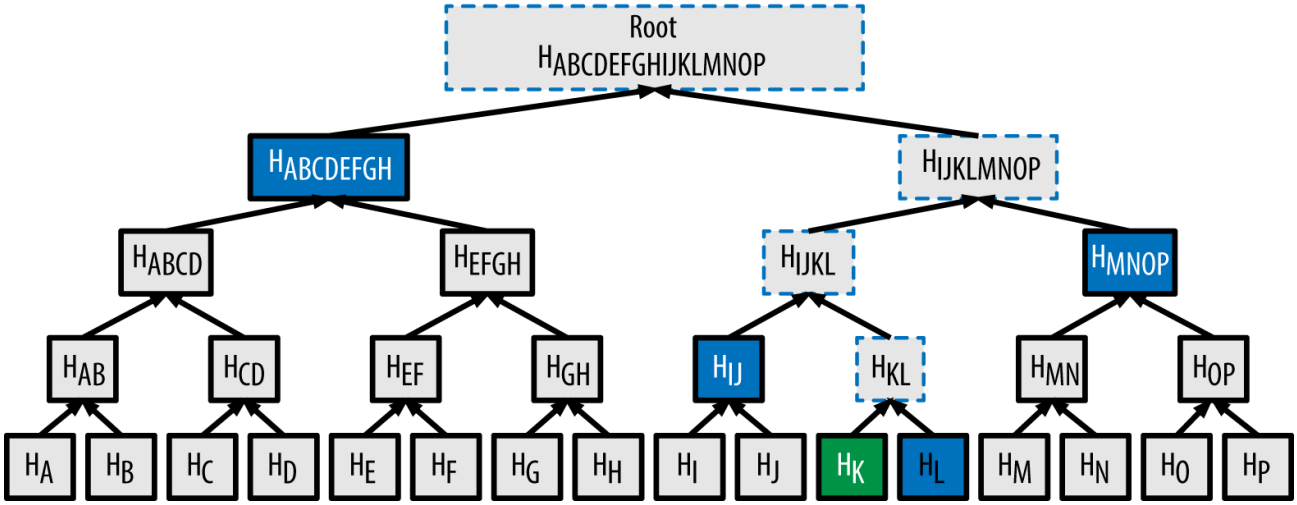


Figure 3.1: Merkle Tree

Chapter 4

Blockchains

4.1 Decentralized Networks

A decentralized network architecture distributes workloads among several machines, instead of relying on a single central server. Contrariwise a centralized network relies on one server as their core distribution center. This fact may lead to lack of security, issues of cost and stability. The solution is to expand the network to multiple, decentralized locations where all computational transactions are made from different servers creating multiple strains for data distribution allowing reliable operation. Essentially, it is an interconnected web of computers all supplying capacity in the form of storage or computing power to assist in a common goal. Expansion of a network across various locations cuts out the necessity of a central server, which enhances security and stability with the added bonus of saving costs[14].

4.2 Blockchain

4.2.1 Blockchain Overview

A blockchain is a decentralized, distributed digital ledger consisting of records called blocks that is used to record transactions across many computers so that any involved block cannot be altered retroactively, without the alteration of all subsequent blocks. Blocks consist of pieces of information regarding transactions like the date, time, and cost of the transaction. Also store information about who is participating in transactions and each a unique code called a "hash" that allows us to tell it apart from every other block. Hashes are cryptographic codes created by special algorithms.

When a block stores new data it is added to the blockchain. Blockchain consists of multiple blocks strung together. In order for a block to be added to the blockchain, however, four things must happen:

• A transaction must occur. In many cases a block will group together many transactions

• That transaction must be verified. After making that purchase, your transaction must be verified. When you make your transaction, a network of computers rushes to check that your transaction happened in the way you said it did. That is, they confirm the details of the purchase, including the transaction's time, dollar amount, and participants.

• That transaction must be stored in a block. After your transaction has been verified as accurate, it gets the green light. The transaction's dollar amount, your digital signature, and the other participant's digital signature are all stored in a block.

• That block must be given a hash. Once all of a block's transactions have been verified, it must be given a unique, identifying code called a hash. The block is also given the hash of the most recent block added to the blockchain. Once hashed, the block can be added to the blockchain. When that new block is added to the blockchain, it becomes publicly available for anyone to view.

Blockchain technology accounts for the issues of security and trust in several ways. First, new blocks are always stored linearly and chronologically. That is, they are always added to the "end" of the blockchain.

After a block has been added to the end of the blockchain, it is very difficult to go back and alter the contents of the block. That's because each block contains its own hash, along with the hash of the

block before it. Hash codes are created by a math function that turns digital information into a string of numbers and letters. If that information is edited in any way, the hash code changes as well. The blocks are connected with hash pointers. Hash Pointer is comprised of two parts:

- Pointer to where some information is stored
- Cryptographic hash of that information

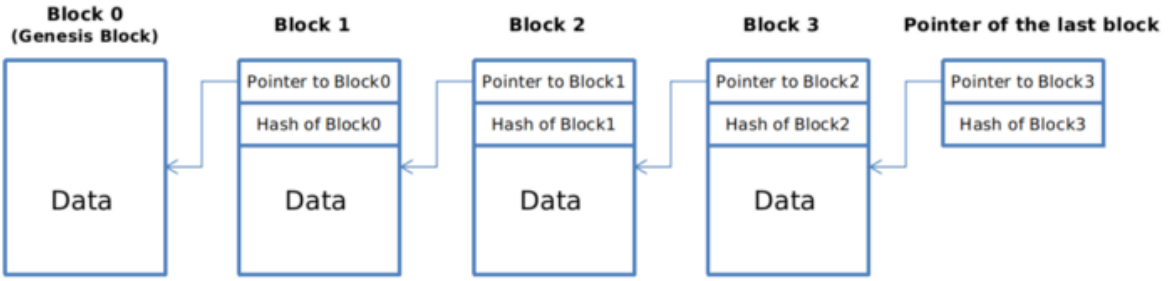


Figure 4.1: Blockchain
https://medium.com/@zhaohuabing/hash-pointers-and-data-structures-f85d5fe91659

The pointer can be used to get the information, the hash can be used to verify that information has not been changed. We should note that the hash stored in the hash pointer is the hash of the whole data of the previous block, which also includes the hash pointer to the block before that one. This makes it's impossible to tamper a block in the blockchain without letting others know. We only need to keep the hash pointer to the last block of the blockchain. Then when somebody shows the whole blockchain later and claim the data in it is not modified, we can tell if any block in the chain is tampered by traversing the blocks backwards and verifying the hashes one by one.

Explanation

- An attacker wants to tamper with one block of the chain, let's say, block 1.
- The attacker changed the content of block 1, because of "collision free" property of the hash function, he is not able to find another data which has the same hash with the old one. So now the hash of this modified block is also changed.
- To avoid others noticing the inconsistency, he also needs to change the hash pointer of that block in the next block, which is block 2.
- Now the content of block 2 is changed, so to make this story consistent, the hash pointer in block3 must be changed.
- Finally, the attacker goes to the hash pointer to the last block of the blockchain, which is a roadblock for him, because we keep and remember that hash pointer.

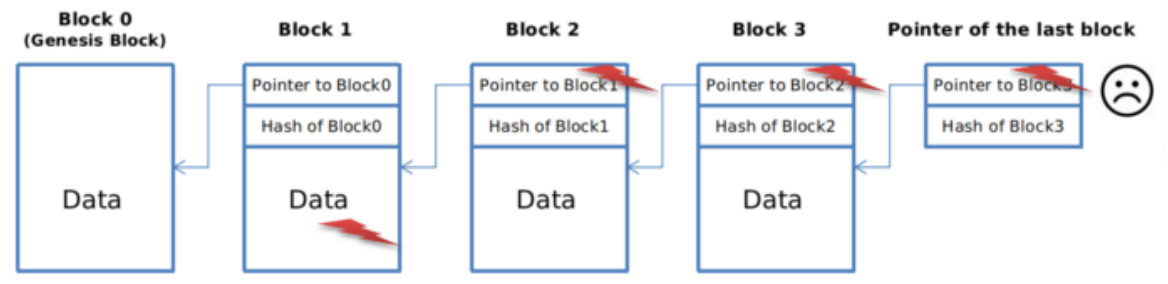


Figure 4.2: Tampered Block
https://medium.com/@zhaohuabing/hash-pointers-and-data-structures-f85d5fe91659

To address the issue of trust, blockchain networks have implemented tests for computers that want to join and add blocks to the chain. The tests, called "consensus models," require users to "prove"

themselves before they can participate in a blockchain network. One of the most common examples is called “proof of work.”

In the proof of work system, computers must “prove” that they have done “work” by solving a complex computational math problem. If a computer solves one of these problems, they become eligible to add a block to the blockchain. But the process of adding blocks to the blockchain, what the cryptocurrency world calls “mining,” is not easy

Proof of work does not make attacks by hackers impossible, but it does make them somewhat useless. If a hacker wanted to coordinate an attack on the blockchain, they would need to control more than 50% of all computing power on the blockchain so as to be able to overwhelm all other participants in the network[13, 15].

4.3 Distributed and Decentralized Networks

A decentralized system is a subset of a distributed system. The primary difference is how/where the “decision” is made and how the information is shared throughout the control nodes in the system. Decentralized means that there is no single point where the decision is made. Every node makes a decision for it’s own behaviour and the resulting system behaviour is the aggregate response.

Distributed means that the processing is shared across multiple nodes, but the decisions may still be centralized and use complete system knowledge.

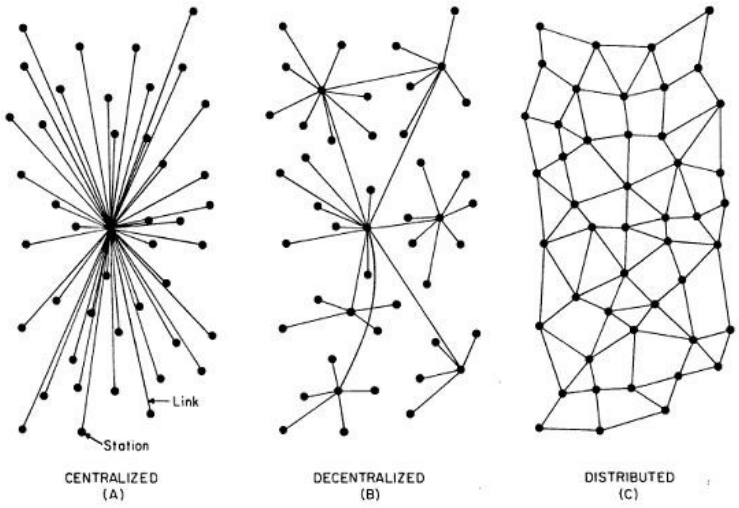


Figure 4.3: Centralized, Decentralizes, Distributed

4.4 Importance and Applications

Security: As mentioned above Blockchain provides one of the most secure way of transactions. Security is the primary concern for all kinds of online activities. Lots of data are stolen, and information is breached in this world of digital. Blockchain provides a very high level of security which makes it impossible to breach for anyone because of the decentralized nature of Blockchain. Due to its high level of security and numerous application, blockchain technology is very flexible in terms of usage. The cryptography used in the Blockchain makes it extremely useful for the execution of the transaction in a very flexible manner.Moreover,over the days there have been numerous hacking and data leaks incident in the past that has shaken the trust of people to keep their data and personal information. But with the use of blockchain technology, Data and information are very much secured, and there is no possibility of any kind of data leaking and hacking.

Speed: The transactions that take place using blockchain technology take very little time to complete.

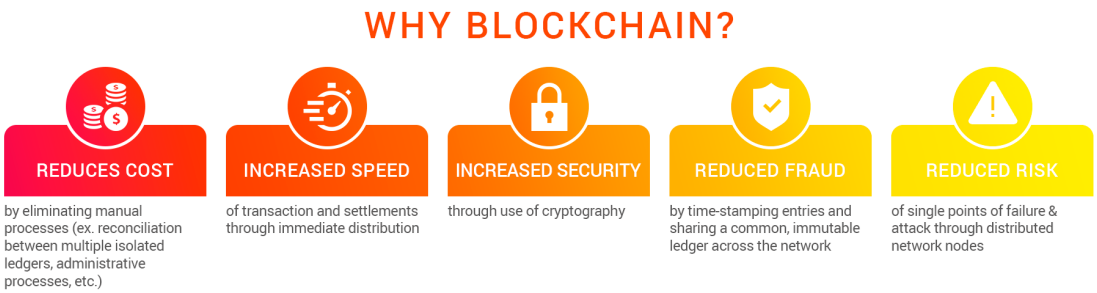


Figure 4.4: Benefits of Blockchain
https://dashbouquet.com/blog/blockchain/how-blockchain-can-transform-artificial-intelligence

It is a lot faster than the transaction time taken in traditional technology. Within a couple of minutes, One can receive or send financial documents and money. There is no burden to wait for hours in this blockchain technology. There is no involvement of any third party in blockchain technology. Thus, it saves a lot of intermediaries cost, and all transactions happen directly from an individual to another individual. In the traditional banking system, the price is more to process financial transactions. Using blockchain technology, banks and companies can increase their economic efficiency.
Reduce cost: Blockchain technology is the most reasonable financial model available right now in the world. If one compares it with traditional economic models, then it is very less expensive. Lots of companies are now looking to use the blockchain technology because they can save lots of cost in their economic model, it is especially beneficial for banking industries.
Transparency: Also, blockchain offers transparency as everything is visible to all the participants from the beginning till date.One can see each and everything on the decentralized network which makes it very open technology. It reduces the chance for any kind of discrepancy in the system because nothing is hidden. Due to the high transparency of transactions in blockchain technology, any kind of fraud can be easily identified [3].

4.4.1 Blockchain in Business

Traditional systems tend to be cumbersome, error-prone and maddeningly slow. Intermediaries are often needed to mediate the process and resolve conflicts. Naturally, this costs stress, time, and money. In contrast, users find the blockchain cheaper, more transparent, and more effective. Small wonder that a growing number of financial services are using this system to introduce innovations, such as smart bonds and smart contracts. The former automatically pays bondholders their coupons once certain preprogrammed terms are met. The latter are digital contracts that self-execute and self-maintain, again when terms are met.

Asset Management: Trade Processing and Settlement Traditional trade processes within asset management (where parties trade and manage assets) can be expensive and risky, particularly when it comes to cross-border transactions. Each party in the process, such as broker, custodian, or the settlement manager, keeps their own records which create significant inefficiencies and room for error. The blockchain ledger reduces error by encrypting the records. At the same time, the ledger simplifies the process, while canceling the need for intermediaries.

Insurance: Claims processing Claims processing can be a frustrating and thankless procedure. Insurance processors have to wade through fraudulent claims, fragmented data sources, or abandoned policies for users to state a few – and process these forms manually. Room for error is huge. The blockchain provides a perfect system for risk-free management and transparency. Its encryption properties allow insurers to capture the ownership of assets to be insured.

Payments: Cross-Border Payments The global payments sector is error-prone, costly, and open to

money laundering. It takes days if not longer for money to cross the world. The blockchain is already providing solutions with remittance companies such as Abra, Align Commerce and Bitspark that offer end-to-end blockchain powered remittance services. In 2004, Santander became one of the first banks to merge blockchain to a payments app, enabling customers to make international payments 24 hours a day, while clearing the next day.

4.4.2 Smart Property

A tangible or intangible property, such as cars, houses, or cookers, on the one hand, or patents, property titles, or company shares, on the other, can have smart technology embedded in them. Such registration can be stored on the ledger along with contractual details of others who are allowed ownership in this property. Smart keys could be used to facilitate access to the permitted party. The ledger stores and allows the exchange of these smart keys once the contract is verified.

The decentralized ledger also becomes a system for recording and managing property rights as well as enabling the smart contracts to be duplicated if records or the smart key is lost.

Making property smart decreases your risks of running into fraud, mediation fees, and questionable business situations. At the same time, it increases trust and efficiency.

Unconventional money lenders/ hard money lending Smart contracts can revolutionize the traditional lending system. For instance, unconventional money lenders (e.g. hard money lenders) service borrowers who have poor credit with needed loans – while charging two to ten percent of the loan amount and claiming their property as collateral. Too many borrowers fall into bankruptcy and lose homes. The blockchain can undercut this by allowing a stranger to loan you money and taking your smart property as collateral. No need to show the lender credit or work history. No need to manually process the numerous documents. The property's encoded on the blockchain for all to see.

Your car/ smartphone Primitive forms of smart property exist. Your car-key, for instance, may be outfitted with an immobilizer, where the car can only be activated once you tap the right protocol on the key. Your smartphone too will only function once you type in the right PIN code. Both work on cryptography to protect your ownership.The problem with primitive forms of smart property is that the key is usually held in a physical container, such as the car key or SIM card, and can't be easily transferred or copied. The blockchain ledger solves this problem by allowing blockchain miners to replace and replicate a lost protocol.

Blockchain Internet-of-Things (IoT) Any material object is a 'thing.' It becomes an internet of things (IoT) when it has an on/ off switch that connects it to the internet and to each other. By being connected to a computer network, the object, such as a car, become more than just an object. It is now people-people, people-things, and things-things. The analyst firm Gartner says that by 2020 there will be over 26 billion connected devices. Others raise that number to over 100!How does the IoT affect you? Your printer can automatically order cartridges from Amazon when it runs low. Your alarm clock will change your time for brewing coffee, while your oven will produce an immaculately timed turkey for Thanksgiving. These are just some examples. On a larger scale, cities and governments can use IoT to develop cleaner environments, more efficient energy use and so-called 'smart cities,' to improve how we live and work.

4.4.3 Smart Contracts

Smart contracts are digital which are embedded with an if-this-then-that (IFTTT) code, which gives them self-execution. In real life, an intermediary ensures that all parties follow through on terms. The blockchain not only waives the need for third parties, but also ensures that all ledger participants know the contract details and that contractual terms implement automatically once conditions are met.

You can use smart contracts for all sort of situations, such as financial derivatives, insurance premiums, property law, and crowd funding agreements, among others.

Blockchain Healthcare Personal health records could be encoded and stored on the blockchain with a private key which would grant access only to specific individuals. The same strategy could be

used to ensure that research is conducted via HIPAA laws (in a secure and confidential way). Receipts of surgeries could be stored on a blockchain and automatically sent to insurance providers as proof-of-delivery. The ledger, too, could be used for general health care management, such as supervising drugs, regulation compliance, testing results, and managing healthcare supplies.

Blockchain music Key problems in the music industry include ownership rights, royalty distribution, and transparency. The digital music industry focuses on monetizing productions, while ownership rights are often overlooked. The blockchain and smart contracts technology can circuit this problem by creating a comprehensive and accurate decentralized database of music rights. At the same time, the ledger and provide transparent transmission of artist royalties and real time distributions to all involved with the labels. Players would be paid with digital currency according to the specified terms of the contract.

Blockchain Government In the 2016 election, Democrats and Republicans questioned the security of the voting system. The Green Party called for a recount in Wisconsin, Pennsylvania, and Michigan. Computer scientists say hackers can rig the electronic system to manipulate votes. The ledger would prevent this since votes become encrypted. Private individuals can confirm that their votes were counted and confirm who they voted for. The system saves money, by the way, for the government, too.The blockchain ledger, also, provides a platform for what we call "responsive, open data." According to a 2013 report from McKinsey and Company, open data – freely accessible government-sourced data that is available over the internet to all citizens – can make the world richer by $2.6 trillion. Startups can use this data to uncover fraudulent schemes, farmers can use it to perform precision farm-cropping, and parents can investigate the side effects of medicine for their sick children. Right now, this data is released only once a year and is, largely, non-responsive to citizens input. The blockchain, as a public ledger, can open this data to citizens whenever and wherever they want. Examples:

Public value/ community The blockchain can facilitate self-organization by providing a self- management platform for companies, NGOs, foundations, government agencies, academics, and individual citizens. Parties can interact and exchange information on a global and transparent scale – think of Google Cloud, but larger and less risky.

Vested responsibility Smart contracts can ensure that electorates can be elected by the people for the people so that government is what it's meant to be. The contracts specify the electorate's expectations and electors will get paid only once they do what the electorate demanded rather than what funders desired.

Blockchain Identity Whether we like it or not, online companies know all about us. Some companies whom we purchase from sell our identity details to advertisers who send you their ads. The blockchain blocks this by creating a protected data point where you encrypt only the information that you want relevant people to know at certain times. For example, if you're going to a bar, the bartender simply needs the information that tells him you're over 18 [2].

4.5 Ethereum

4.5.1 Ethereum Introduction

Overview

Ethereum is an open source, globally decentralized computing infrastructure that executes problems called smart-contracts. It uses a blockchain to synchronize and store the system's state changes, along with a cryptocurrancy called ether to meter and constrain execution resource costs.

Ethereum purpose is not primarily to be a digital currency payment method. Ether is intended as a utility currency to pay for use of the Ethereum platform as the world computer. Ehereum is meant to be a blockchain without a specific purpose, that could support a broad variety of applications by being programmed.

Turing Completeness

Ethereum is a Turing Complete system that can be used to simulate any Turing Machine. Turing Complete means that any problem of any complexity can be computed by ethereum. Turing Complete systems can run infinite loops (can arise by accident), that means that a smart contract can run forever.

Metering Mechanism

The metering mechanism being used is called gas. Every instruction has a predetermined cost in units of gas. When a transaction triggers the execution of a smart contract it must include a number of gas that sets the upper limit of what can be consumed by running the smart contract. Gas is the mechanism Ethereum uses to allow Turing Complete computation while limiting the resources that any program can consume. Ether needs to be sent in every transaction and it needs to be explicitly earmarked for the purchase of gas, along with an acceptable gas price. Gas is purchased for the execution of the transaction and any unused gas is refunded back to the sender of the transaction.

Wallet

A software application that helps you manage your Ethereum account. It holds your keys an can create and broadcast transactions in your behalf.

EVM

Ether is meant to be used to pay for the execution of smart contracts which are computer programs that run on an emulated computer called the EVM. EVM is global singleton, meaning that it operates as if it were a global, single instance computer running everywhere. Each node on the Ethereum network runs a copy of the EVM to validate contract execution.

Accounts

In Ethereum there are two types of accounts. The first one is the Externally Owned Account, that have a private key means control over access to funds and contracts (wallet). The second type is the contract account. A contract account has smart contract code. It doesn't have a private key. Instead it is owned and controlled by the logic of its smart contract code. Contracts have addresses, can also send and receive ether. When a transaction's destination is contract's address, it causes the contract to run, using the transaction's data as input.

Clients

It is a software application that implements the Ethereum specification and communicates over peer-to-peer network with other Ethereum clients.

4.5.2 Keys and Addresses

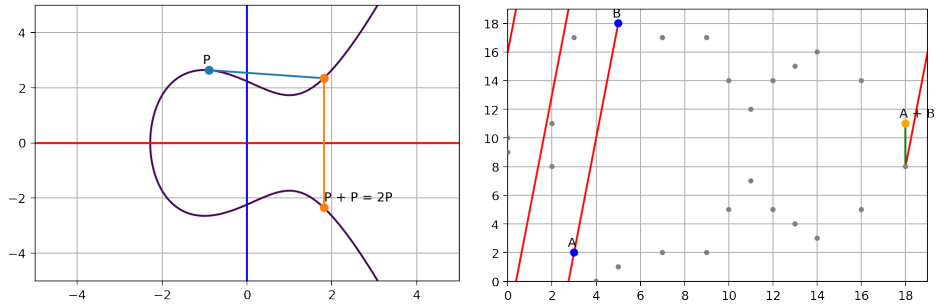
A private key uniquely determines a single Ethereum address, also known as an account. Private key must remain private and never be reveled, stored or transmitted on Ethereum. Access and control of funds is achieved with digital signatures, which are also created using the private key. Ethereum transactions require a valid digital signature to be included in the blockchain. The digital signatures in Ethereum transactions prove the owner of the funds, because they prove ownership of the private key. In public-key cryptography-based systems, such as Ethereum, keys come in pairs consisting of a private key and a public key. Ethereum uses the Elliptic curve cryptography:

- Private key: completely random

- Public key: point on the elliptic curve, it is a set (x,y) coordinates that satisfy the elliptic curve equation (two numbers joined together). Public key is generated from private key using ”only go one way” calculation called trapdoor function. Trapdoor function is a function that is easy to compute in one direction, yet difficult to compute in the opposite side, without special information call trapdoor.

In Elliptic curve a point G can be multiplied by an integer k to produce another point K. But there is no such thing as division.

Example: elliptic curve y2modp = (x3 + 7)modp over a finite-field (we add the mod because we can’t use real numbers due to computational problems), where p is a prime large number.



(a) Elliptic curve real numbers (b) Elliptic curve over finite field

Figure 4.5: Elliptic curve

As we see on figure (a) in order to take the point 2P, we extend the tangent line of our current position P until it hits the curve then we take the reflection of that point in respect to x-axis. That is the simple case. So if i have a random point NP, the only way to find the N is brute force.

If we have a slightly bigger number, say 227P, let’s first break it down to binary so that we can get its power of two composition. Its binary representation is:

11100011

in other words, summing up all the power of two values would be
2^7P+2^6P+2^5P+2^1P+2^0P
128P+64P+32P+2P+P So the operation we need would be

- Add P and double P
- Add 2P and double 2P
- Double 4P
- Double 8P
- Double 16P
- Add 32P and double 32P
- Add 64P and double 64P
- Add 128P and double 128P

That’s only 8 steps, compares to 227 steps, it’s way faster. This method is called double and add. It allows us to get to the desired multiple times of point jumping on an elliptic curve pretty fast. The number 227 in this example is small, given that we can approach the number we want with O(logn).

When I have a finite-field modulo p the line when it hits the boundaries, it actually wrap-around to the other end, as the modulo operation is circling around as we see above at the figure (b) [6].

Generating a public key: starting with the private key k, we multiply it by a predetermined point of the curve called generator point to produce another point somewhere else on the curve, which is the corresponding public key K=k*G (K = (x,y)). The serialisation concatenates x,y with prefix 04.

Ethereum Addresses are unique identifiers that are derived from public keys or contracts using a one way hash function. Ethereum uses the Keccak-256 function to calculate the hash of the public key and keeps only the last 20 bytes that consist the address.

4.5.3 Wallet Technology Overview

One key consideration in designing wallets is balancing convenience and privacy. The most convenient ethereum wallet is one with a single private key and address that you reuse for everything. This is a privacy nightmare, as anyone can easily track and correlate all your transactions. Using new key for every transaction is best for privacy, but becomes very difficult to manage. The wallet holds only keys. The ether or other token are recorded on the Ethereum blockchain.

There are two primary types of wallets:

- The first type is a non-deterministic wallet, where each key is independently generated from a different random number. These keys are non related to each other. These keys are also known as JBOK wallets.
- The second type is a deterministic wallet where all the keys are derived from a single master key, known as seed. All the keys are related to each other and can be generated if you have the original seed. To make them more secure against data-loss accidents, the seeds are often encoded as a list of words.

Hierarchical Deterministic Wallets

Contains keys derived in a tree structure, such that a parent key can a parent key can derive a sequence of child keys.

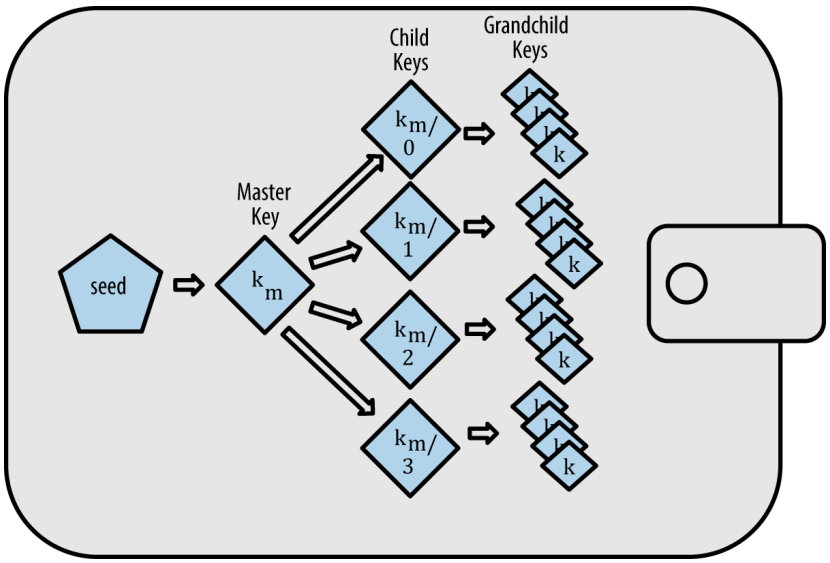


Figure 4.6: Hierarchical Deterministic Wallet

Advantage of Hierarchical Deterministic wallet: users can create a sub sequence of public keys without having access to the corresponding private keys. This allows hierarchical deterministic wallets to be used on an insecure server.

Generating mnemonic words

Mnemonic code words are word sequences that encode a random number used as a seed to derive a deterministic wallet. The sequence of words is sufficient to recreate the seed, and from there recreate the wallet and all the derived keys. A wallet application that implements deterministic wallets with mnemonic words will show the user a sequence of 12 to 24 words when first creating a wallet. That sequence of words is the wallet backup, and can be used to recover and recreate all the keys in the same or any compatible wallet application. The procedure is:

1. Create a cryptographical random sequence S of 128 to 256 bits.
2. Create a checksum of S by taking the first length-of-S / 32 bits of the SHA-256 hash of S.
3. Add the checksum to the end of the random sequence S.
4. Divide the sequence-and-checksum concatenation into subsections of 11 bits.
5. Map each 11-bit value to a word from the predefined dictionary of 2,048 words.
6. Create the mnemonic code from the sequence of words, maintaining the order.

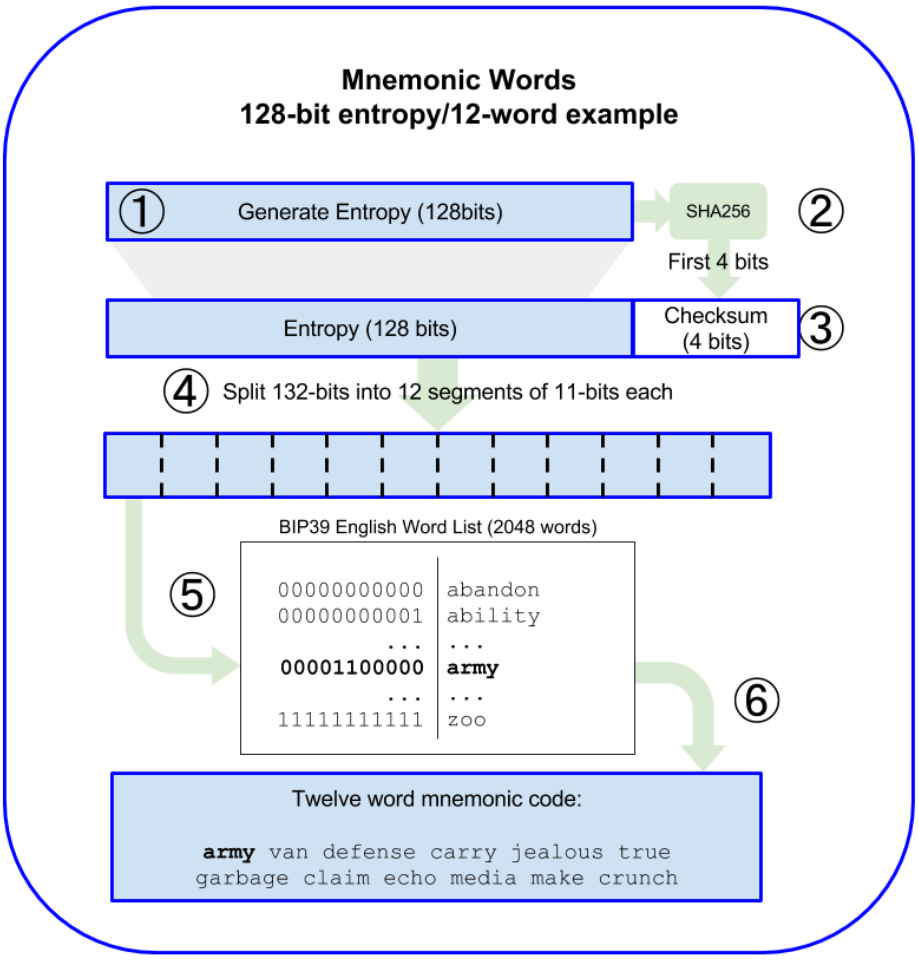


Figure 4.7: Mnemonic Words

The entropy is then used to derive a longer (512-bit) seed through the use of the key-stretching function. The key-stretching function takes two parameters: the mnemonic and a salt. The purpose of a salt in a key-stretching function is to make it difficult to build a lookup table enabling a brute-force attack.

7. The first parameter to the key-stretching function is the mnemonic produced in step 6.
8. The second parameter to the key-stretching function is a salt. The salt is composed of the string constant ”mnemonic” concatenated with an optional user-supplied passphrase.
9. PBKDF2 (the key-stretching function) stretches the mnemonic and salt parameters using 2,048 rounds of hashing with the HMAC-SHA512 algorithm, producing a 512-bit value as its final output. That 512-bit value is the seed.

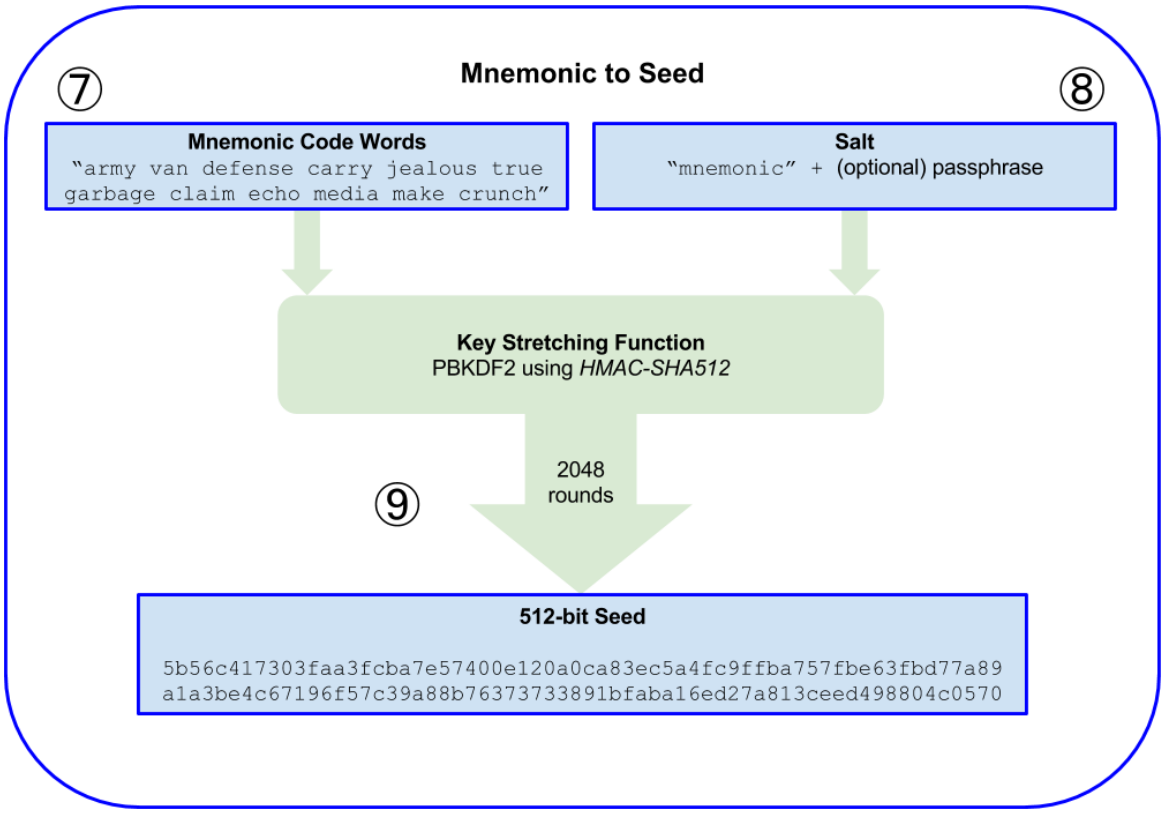


Figure 4.8: Mnemonic to Seed

Extended public and private keys

These keys san be used to derive ”child” keys. Extending a key involves taking the key itself and appending a special chain code to it. A chain code is a 256-bit binary string that is mixed with each key to produce child keys.

There are 2 ways to produce a public key, either directly the child private key, or from the parent public key.

Hardened child key derivation

The ability to derive a branch of public keys from an extended public key, or xpub, is very useful, but it comes with a potential risk. Access to an xpub does not give access to child private keys. However,

because the xpub contains the chain code (used to derive child public keys from the parent public key), if a child private key is known, or somehow leaked, it can be used with the chain code to derive all the other child private keys. A single leaked child private key, together with a parent chain code, reveals all the private keys of all the children. Worse, the child private key together with a parent chain code can be used to deduce the parent private key.

To counter this risk, HD wallets use an alternative derivation function called hardened derivation, which ”breaks” the relationship between parent public key and child chain code. The hardened derivation function uses the parent private key to derive the child chain code, instead of the parent public key. This creates a ”firewall” in the parent/child sequence, with a chain code that cannot be used to compromise a parent or sibling private key.

In simple terms, if you want to use the convenience of an xpub to derive branches of public keys without exposing yourself to the risk of a leaked chain code, you should derive it from a hardened parent, rather than a normal parent. Best practice is to have the level-1 children of the master keys always derived by hardened derivation, to prevent compromise of the master keys.

Index numbers for normal and hardened derivation

It is clearly desirable to be able to derive more than one child key from a given parent key. To manage this, an index number is used. Each index number, when combined with a parent key using the special child derivation function, gives a different child key. The index number used in the BIP-32 parent-to-child derivation function is a 32-bit integer. To easily distinguish between keys derived through the normal (unhardened) derivation function versus keys derived through hardened derivation, this index number is split into two ranges. Index numbers between 0 and 2^{31} - 1 (0x0 to 0x7FFFFFFF) are used only for normal derivation. Index numbers between 2^{31} and 2^{32} - 1 (0x80000000 to 0xFFFFFFFF) are used only for hardened derivation. Therefore, if the index number is less than 2^{31}, the child is normal, whereas if the index number is equal to or above 2^{31}, the child is hardened.

To make the index numbers easier to read and display, the index numbers for hardened children are displayed starting from zero, but with a prime symbol. The first normal child key is therefore displayed as 0, whereas the first hardened child (index 0x80000000) is displayed as 0&#x27.

In sequence, then, the second hardened key would have index of 0x80000001 and would be displayed as 1&#x27, and so on. When you see an HD wallet index i&#x27, that means 2^{31} + i.

The HD wallet key identifier is: The ”ancestry” of a key is read from right to left, until you reach the master key from which it was derived. For example, identifier m/x/y/z describes the key that is the z-th child of key m/x/y, which is the y-th child of key m/x, which is the x-th child of m.

Navigating the HD wallet tree structure

Extending that specification, BIP-44 proposes a multicurrency multiaccount structure signified by setting the ”purpose” number to 44’. All HD wallets following the BIP-44 structure are identified by the fact that they only use one branch of the tree: m/44’/*.

BIP-44 specifies the structure as consisting of five predefined tree levels:

m / purpose / cointype / account / change / addressndex

The first level, purpose', is always set to 44'. The second level, cointype', specifies the type of cryptocurrency coin, allowing for multicurrency HD wallets where each currency has its own subtree under the second level. There are several currencies defined in a standards document called SLIP0044; for example, Ethereum is m/44'/60', Ethereum Classic is m/44'/61', Bitcoin is m/44'/0', and Testnet for all currencies is m/44'/1'.

The third level of the tree is account', which allows users to subdivide their wallets into separate logical subaccounts for accounting or organizational purposes. For example, an HD wallet might contain two Ethereum ”accounts”: m/44'/60'/0' and m/44'/60'/1'. Each account is the root of its own subtree.

Because BIP-44 was created originally for Bitcoin, it contains a ”quirk” that isn’t relevant in the Ethereum world. On the fourth level of the path, change, an HD wallet has two subtrees: one for creating receiving addresses and one for creating change addresses. Only the ”receive” path is used in Ethereum, as there is no necessity for a change address.

Note that whereas the previous levels used hardened derivation, this level uses normal derivation. This is to allow the account level of the tree to export extended public keys for use in a nonsecured environment. Usable addresses are derived by the HD wallet as children of the fourth level, making the fifth level of the tree the addressindex. For example, the third receiving address for Ethereum payments in the primary account would be M/44'/60'/0'/0/2.

HD path	Key described
M/44'/60'/0'/0/2	The third receiving public key for the primary Ethereum account7
M/44'/0'/3'/1/14	The 15th change-address public key for the 4th Bitcoin account
m/44'/2'/0'/0/1	The second private key in the Litecoin main account, for signing transactions

4.5.4 Transactions

Transactions are signed messages originated by an externally owned account, transmitted by the Ethereum network, and recorded on the Ethereum blockchain.

The structure of a Transaction

A transaction is a serialized binary message that contains the following data:

Nonce:
A sequence number, issued by the originating EOA, used to prevent message replay

Gas price:
The price of gas (in wei) the originator is willing to pay

Gas limit:
The maximum amount of gas the originator is willing to buy for this transaction

Recipient:
The destination Ethereum address

Value:
The amount of ether to send to the destination

Data:
The variable-length binary data payload

v,r,s:
The three components of an ECDSA digital signature of the originating EOA

The transaction message’s structure is serialized using the Recursive Length Prefix (RLP) encoding scheme, which was created specifically for simple, byte-perfect data serialization in Ethereum. While this is the actual transaction structure transmitted, most internal representations and user interface visualizations embellish this with additional information, derived from the transaction or from the blockchain. For example, you may notice there is no “from” data in the address identifying the originator EOA. That is because the EOA’s public key can be derived from the v,r,s components of the ECDSA signature. The address can, in turn, be derived from the public key. When you see a transaction showing a ”from” field, that was added by the software used to visualize the transaction. Other metadata frequently added to the transaction by client software includes the block number (once it is mined and included in the blockchain) and a transaction ID (calculated hash). Again, this data is derived from the transaction, and does not form part of the transaction message itself.

Nonce

The nonce is one of the most important and least understood components of a transaction. The definition in the Yellow Paper:

Nonce: A scalar value equal to the number of transactions sent from this address or, in the case of accounts with associated code, the number of contract-creations made by this account.

There are two scenarios where the existence of a transaction-counting nonce is important: the usability feature of transactions being included in the order of creation, and the vital feature of transaction duplication protection. Let's look at an example scenario for each of these:

- Imagine you wish to make two transactions. You have an important payment to make of 6 ether, and also another payment of 8 ether. You sign and broadcast the 6-ether transaction first, because it is the more important one, and then you sign and broadcast the second, 8-ether transaction. Sadly, you have overlooked the fact that your account contains only 10 ether, so the network can't accept both transactions: one of them will fail. Because you sent the more important 6-ether one first, you understandably expect that one to go through and the 8-ether one to be rejected. However, in a decentralized system like Ethereum, nodes may receive the transactions in either order; there is no guarantee that a particular node will have one transaction propagated to it before the other. As such, it will almost certainly be the case that some nodes receive the 6-ether transaction first and others receive the 8-ether transaction first. Without the nonce, it would be random as to which one gets accepted and which rejected. However, with the nonce included, the first transaction you sent will have a nonce of, let's say, 3, while the 8-ether transaction has the next nonce value (i.e., 4). So, that transaction will be ignored until the transactions with nonces from 0 to 3 have been processed, even if it is received first. Phew!

- Now imagine you have an account with 100 ether. Fantastic! You find someone online who will accept payment in ether for a mcguffin-widget that you really want to buy. You send them 2 ether and they send you the mcguffin-widget. Lovely. To make that 2-ether payment, you signed a transaction sending 2 ether from your account to their account, and then broadcast it to the Ethereum network to be verified and included on the blockchain. Now, without a nonce value in the transaction, a second transaction sending 2 ether to the same address a second time will look exactly the same as the first transaction. This means that anyone who sees your transaction on the Ethereum network (which means everyone, including the recipient or your enemies) can "replay" the transaction again and again and again until all your ether is gone simply by copying and pasting your original transaction and resending it to the network. However, with the nonce value included in the transaction data, every single transaction is unique, even when sending the same amount of ether to the same recipient address multiple times. Thus, by having the incrementing nonce as part of the transaction, it is simply not possible for anyone to "duplicate" a payment you have made.

The Ethereum network processes transactions sequentially, based on the nonce. That means that if you transmit a transaction with nonce 0 and then transmit a transaction with nonce 2, the second transaction will not be included in any blocks. It will be stored in the mempool, while the Ethereum network waits for the missing nonce to appear. Ethereum also allows concurrency of operations but enforces a singleton state through consensus.

Transaction gas

Gas is the fuel of Ethereum, it uses gas to control the amount of resources that a transaction can use. The gas price field in the transaction allows the originator to set the price they are willing to pay in exchange for gas.

Wallets can adjust the gas price in the transactions they originate to achieve faster confirmation of transactions. The higher the gas price, the faster the transaction is to be confirmed. During periods of

low-demand for space in block transaction with gas price set to zero can as well be mined. Gaslimit is the maximum number of gas units the originator is willing to buy.

Transaction Recipient

Ethereum doesn't validate recipient address in transactions. You can send to an address that has no corresponding private key or contract, thereby "burning" the ether, rendering it forever unspendable.

Transaction Value and Data

Transaction can have both value and data, only value, only data or neither of them. A transaction with only value is a payment. A transaction with only data is an invocation.

Transmitting value to EOAs and contracts:

- EOA: payment. Ethereum will record a state change adding the value you sent to the balance of the address
- Contract: EVM will execute the contract and will attempt to call the function named in the data payload of your transaction. If there is no data, the EVM will call a fallback function and if that function is payable, will execute it to know what to do next. If there is no code in the fallback function, then the balance of the contract will be increased. If there is no fallback function or non-payable fallback function the transaction will be reverted.

Transmitting data payload to an EOA or contract:

- EOA: If you send data to an EOA the interpretation of the data is up to the wallet you use to access the EOA.
- Contract: Most contracts use the data as a function invocation, calling the named function and passing any encoded arguments to the function.

The data payload sent to a contract is a hex-serialized encoding of:

- A function selector: The first 4-bytes of Keccak-256 hash of the function's prototype. The prototype is defined as the string containing the name of the function, followed by the data-types of each of the arguments enclosed in parentheses and separates with commas.
- The function's arguments encoded according to the rules of the various elementary types (API).

There is a special transaction named contract creation. This transaction is being sent to a special destination called zero address. It can never spend ether or initiate a transaction. Any ether sent to 0x0 will become unspendable and be lost forever. A contract creation transaction needs only to contain a data payload that contains the compiled bytecode which will create the contract. If you include ether the new contract will be set up with a starting balance.

4.5.5 Digital signature

Wikipedia's Definition of a Digital Signature A digital signature is a mathematical scheme for presenting the authenticity of digital messages or documents. A valid digital signature gives a recipient reason to believe that the message was created by a known sender (authentication), that the sender cannot deny having sent the message (non-repudiation), and that the message was not altered in transit (integrity) [8].

The Digital signature:

- proves that the owner of the private key, has authorized the transaction
- It guarantees non-repudiation.

- proves that the transaction's data cannot be modified by anyone after the transaction has been signed.

Digital signature consists of two parts: 1. algorithm for creating the signature using the private key. 2. algorithm that allows anyone to verify the signature by only using the message and the public key.

$$Sig = sig\left(keccak256\left(m\right), k\right)$$
$$Sig = (r, s)$$

In order, to verify the signature one must have the signature (r,s), the serialized transaction and the public key, that corresponds to the private key used to sign this transaction. Also, it is important for security reasons to separate the sign and transmit operations. When you sign you must have access to the private keys. Is more secure this process to be executed offline.

4.5.6 Smart Contracts

Smart contracts are simply computer programs. The word contract has no legal meaning in this context. Once the smart contract is deployed, the code of it cannot change. Unlike with traditional software, the only way to modify a new contract is to deploy a new one. The outcome of the execution of a smart contract is the same for everyone who runs it. The EVM runs as a local instance on every Ethereum node, but because all instances of the EVM operate on the same initial state and produce the same final state, the system operates as a single "world computer".

The Ethereum contract ABI

The application Binary Interface is an interface between two program modules; often between the operating system and the user programs. An ABI defines how data structures and functions are accessed in machine code; this is not to be confused with an API, which defines this access in high-level, often human readable codes (source codes).

In Ethereum the ABI is used to encode contract calls for the EVM and to read data out of a transaction. The purpose of an ABI is to define the functions in the contract that can be invoked and describe how each function will accept arguments and return its result. A contract ABI is specified as a JSON array of function description and events. Using the ABI an application can construct transactions and call the functions of a contract using the correct arguments.

Contract Security

In order to keep your contracts safe from attackers you should use defensive programming. Minimalism and simplicity are very important. Complexity is the enemy of security. Also, reuse code, follow the DRY principal (don't repeat yourself).

4.5.7 Tokens

It is commonly used to refer to a privately issued special-purpose coin-like items of a significant intrinsic value. On blockchain are redefining the word to mean blockchain-based abstractions that can be owned and that represent assets, currency or access rights.

How tokens are used

The most obvious use of the tokens is as digital private currencies. This is only one possible use. Tokens can be programmed to serve many different functions. A token can simultaneously convey a voting right, an access right and ownership of a resource.

Fungibility: Is a property of a good or a commodity whose individual units are essentially interchangeable.

Tokens are fungible when we can substitute any single unit of the token for another without any difference in its value or function.

Non-fungible tokens that each represent a unique tangible or intangible item and therefore are not interchangeable. Each non-fungible token is associated with a unique identifier, such as a serial number.

Counterparty risk: Is the risk that the other party in the transaction will fail to meet their obligations.

Same tokens represent digital items that are intrinsic to the blockchain. Those digital assets are governed by consensus rules, just like the tokens themselves. Tokens that represent intrinsic assets do not carry additional counterparty risk. The blockchain consensus rules apply and your ownership of the private key is equivalent to ownership of the asset, without any intermediary.

Many tokens are used to represent extrinsic things, the ownership of these items that are not ”within” the blockchain are not governed by consensus rules. As the result they carry additional counterparty risk.

One of the most important ramifications of the blockchain based tokens is the ability to convert extrinsic assets into intrinsic and thereby remove counterparty risk.

Equity or Utility

In principal, the use of tokens can be seen as the ultimate organization tool. Utility tokens are those where the use of a token is required to gain access to a service, app or resource. Equity Tokens are those that represents shares in the control or ownership of something such as a startup.

Token standard

In order to create a new token in Ethereum you must create a ne smart contract. The vast majority of tokens are based on ERC20 token standard. The ERC20 standard defines a common interface for contracts implementing a token, such that any compatible token can be assessed and used it the same way. The interface consists of a number of functions that must be present in every implementation of the standard, as well as some optional functions and attributes that may be added by developers.

4.5.8 Oracles

Oracles are systems that can provide external data sources to the Ethereum smart contracts. In order to maintain consensus EVM execution must be totally deterministic and based only on the shared context of the Ethereum state and signed transactions.

The execution of a random function is prohibited. Consider the effect to achieve consensus after the execution of such a function: node A would execute the command and store 3, while node B would store 7 instead. Oracles, ideally provides a trustless way of getting extrinsic information such as random numbers or arbitrary computation.

Oracles Design Patterns

- Collect data from an off-chain source
- Transfer the data online with a signed message.
- Make the data available by putting it in a smart contract’s storage.

4.5.9 Decentralized Applications (DApps)

Smart Contracts are a way to decentralize the controlling logic and payment functions of applications. Web3 DApps are about decentralizing all other aspects of application: storage, messaging, naming etc A DApp is an application that is mostly or entirely decentralized. Advantages of creating a DApp:

Resiliency:

Because the business logic is controlled by a smart contract, a DApp backend will be fully distributes and managed on a blockchain platform. Unlike an application deployed on a centralized server, a DApp will not have a downtime and will continue to be available as long as the platform is still operating.

Transparency:

The on-chain nature of a DApp allows everyone to inspect the code and be sure about its function. Any interaction with the DApp will be stored forever in the blockchain.

Censorship resistance:

As long as all users have access to an Ethereum node (running one if necessary), the users will always be able to interact with a DApp without interface from any centralized control. No service provider, or even the owner of a smart contract, cal alter the code once it is deployed on the system.

Backend (smart contract):

In a DApp, smart contracts are used to store the business logic (program code) and the related state of your application. You can think of a smart contract replacing a server side (aka backend) component in a regular app.

Data storage

Due to high gas cost and the currently low block gas limit, smart contracts are not well-suited to storing or processing large amount of data. Hence, most DApps utilize off-chain data storage services. That data storage platform can be centralized (for example cloud database), or the data can be decentralized stored on a P2P platform as IPFS, or Ethereum's own Sworm Platform.

4.5.10 The Ethereum Virtual Machine

The EVM is part of the Ethereum that handles smart contract deployment and execution. At a high level, the EVM running on the Ethereum Blockchain thought as a global decentralized computer containing millions of executable objects, each with its own permanent data store.

The EVM is a "quasi"-Turing-Complete state machine; "quasi" because all execution processes are limited to a finite number of computational steps by the gas available for any given smart contract execution. As such, the halting problem is solved and the situation where execution might run forever, thus bringing the Ethereum platform to halt in its entirely, is avoided.

The EVM has a stack-based architecture, storing all in-memory values to the stack. It works with a word size of 256 bit and has addressable data components:

- An immutable program code ROM, loaded with the bytecode of the smart contract to be executed
- A volatile memory, with every location explicitly initiated to zero.
- A permanent storage that is a part of the Ethereum state, also zero initiated.

The EVM Instruction set

- Arithmetic and bitwize logic operations
- Execution context inquiries
- Stack, memory and storage access

- Control flow operations
- Logging, calling and other operators

Ethereum State

The job of the EVM is to update the Ethereum state by computing valid state transitions as a result of smart contract code execution. This aspect leads to the description of Ethereum as a transition-based state machine, which reflects the fact that external actors initiate state transitions by creating, accepting and ordering transactions.

At the top level, we have the Ethereum world state. The world state is a mapping of Ethereum Address (160 bit) to accounts. At the lower level, each Ethereum Address represents an account comprising an ether balance, a nonce, the account's program code.

When a transition result is a smart contract execution an EVM is initiated with all the information required in relation to the current block being created and the specific transaction being processed. In particular, the EVM's program code ROM is loaded, the code of the contract account being called, the program counter is set to zero, the storage is loaded from the contract's account storage, the memory is set to all zeros and all the block and environment variables are set. If the gas is reduced to zero, we get an Out Of Gas exception, the transaction is abandoned. No changes at the Ethereum state are applied.

Gas Counting During Execution

Every EVM opcode has a cost in gas, and so the EVM's gas supply is reduced as the EVM steps through the program. Before every operation, the EVM checks that there is enough gas to pay for the operation's execution. If there isn't enough gas the execution is halted and the transaction is reverted.

When constructing a new block, miners on the Ethereum network can choose among pending transactions by selecting those that offer to pay a higher gas price.

Chapter 5

System's Architecture

5.1 Current HDFS Architecture

Users of HDFS expect that no data will be lost or corrupt during storage and processing. Since Hadoop is handling a huge amount of data the change of data corruption occurring is really high during data flowing through the system. The usual way of detecting corrupted data is by computing the checksum of the data when it first enters the system and again every time it is transmitted through the unreliable channel. This technique overs only error detection through transmission, it doesn't offer any way of fixing the data. It is also possible the checksum to be corrupted and not the data, but very unlikely, because checksum is much smaller than the data.

HDFS transparently checksums all data written to it and by default verifies checksums during read operation. By default checksums are 512 bytes long. Datanodes are responsible for verifying the data they receive before storing it, either they receive the data from the client of other Datanode. A client writing data sends it to a pipeline of datanodes, as explained above and the last datanode in the pipeline verifies the checksum. If it detects an error, the client receives a ChecksumException, a subclass of IOException, which it should handle in an application-specific manner; for example, by retrying the operation.

When clients read data from datanodes, they verify checksums as well, comparing them with the ones stored at the datanode. Each datanode keeps a persistent log of checksum verification, so it knows the last time each of its blocks was verified. When a client successfully verifies a block, it tells the datanode, which updates its log. Keeping statistics such as these is valuable in detecting bad disks.

Aside from block verification on client reads, each datanode runs a DataBlockScanner in a background thread that periodically verifies all the blocks stored on the datanode.

Because HDFS stores replicas of blocks, it can "heal" corrupted blocks by copying one of the good replicas to produce a new, uncorrupt replica. The way this works is that if a client detects an error when reading a block, it reports the bad block and the datanode it was trying to read from to the namenode before throwing a ChecksumException. The namenode marks the block replica as corrupt so it doesn't direct clients to it or try to copy this replica to another datanode. It then schedules a copy of the block to be replicated on another datanode, so its replication factor is back at the expected level. Once this has happened, the corrupt replica is deleted [25].

Another mechanism for finding the corrupted blocks is block report. Interval of block reports is determined by configuration dfs.blockreport.intervalMsec (in hdfs-site.xml). By default this is set to 21600000 milliseconds (6 hours).

Some of the information contained in the block report is: Data node registration, information about the blocks, which contains: block ID, block length, block generation timestamp, state of the block replica (For e.g. replica is finalized or waiting to be recovered etc.)

This information is used by the Name Node for:Process first block report: If it is a first time report for the newly registered Data Node, it just adds all the valid replicas. It ignores all the invalid blocks, till the next block report.For updating the information about blocks: The (Data Node -> Blocks) map is updated in the Name Node. The new block report is compared with the old report and information about successful blocks, corrupted blocks, invalidated blocks etc.

5.2 Corrupted Nodes

The above is a great way to detect errors during data transfer, but it also implies that every node of the Hadoop cluster is reliable. What happens if a node wants to delete the client's data for it's own benefit? It is possible to delete or change the data and the checksums stored in the Datanode, as long as all this information is stored in the Datanode's disk. So the DataBlockScanner will not be able to detect the corruption and inform the Namenode. Even with the block report some of the data may be changed without being perceived. The Datanode may have deleted the block's data, but still sending the report containing the block metadata to the Namenode. Also Namenode may be malicious and collaborate with the corrupted Datanode and verify the corrupted blocks as valid. There are some applications including CloudAgora [7], where Datanodes or even Namenode can't be trusted.

Let's take a look in CloudAgora. It is a decentralized cloud that allows for on-demand and low-cost access to storage and computing infrastructures. The goal of CloudAgora is to create a blockchain-based platform where participants can act either as providers, offering idle CPU and available storage, or as consumers, renting the offered resources and creating ad-hoc virtual cloud infrastructures. Storage and processing capacities are monetized and their prices are governed by the laws of supply and demand [7]. In this application, it is very likely some of the nodes to be corrupted in order to earn more money. If a node doesn't have enough storage to store more data it loses the opportunity to serve a client's request.

The new system can assume that the Namenode is either trusted or not as well, we will examine these two possibilities later. Taking advantage of blockchain's security we managed to present a system much more secure, in order to prevent data corruption in HDFS.

5.3 Explanation of the System

We can explain the system in two parts. One regarding the Merkle Trees and the other the Blockchain, but these components cooperate in order to achieve the security and integrity of the data.

5.3.1 Writing a File

At first we are going to present the process of writing a file on HDFS and the use of Merkle Trees in our system. The initiation of writing a file comes from the Client, when he/she wants to upload a file on HDFS. The file is being separated into HDFS Blocks. Every HDFS Block consist a Merkle Tree. The leafs of the tree are the hashes of the data of the HDFS Block. When a Client completes an HDFS Block the Merkle Tree of the Block is created and the root and the leafs of the tree are being transmitted to the Namenode. The procedure is being explained on the section below.

As said above a Merkle tree is a tree in which every leaf node is labelled with the cryptographic hash of a data block, and every non-leaf node is labelled with the cryptographic hash in the labels of its child nodes. Hash trees allow efficient and secure verification of the contents of large data structures.

When a client writes data a DFSOutputStream caches the file data into a temporary local file. Application writes are transparently redirected to this temporary local file. We added a buffer that also accumulates these data and when it stores data equal to an HDFS block size (128 MB by default), this buffer is redirected to a class named MakeMerkleTree.java. The data are being separated into chunks equal to Merkle Chunk size (as we will see below we chose it to be 10000 bytes). Every Merkle Chunk is being hashed according to SHA256 cryptographic function, every hash is a leaf of the tree called proof. Subsequently, the tree is being created starting from the leafs retrospectively.

Once the tree is created, Datastreamer.java sets to the current Block (block class consists of information about the block that send to the Namenode, it doesn't include the data, it includes the blockid -a long unique for every block, the generation stamp of the block and the bytes) the fields roothash and myproof, which is the hash of the root node and a list consisting of all the hashes of the leafs. Subsequently, using the addBlock method sends this Block to the Namenode. The data are being sent

using Protocol buffers, that are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages.

The Namenode receives the data and stores the list of the proofs in a Rockdb. RocksDB is a high performance embedded database for key-value data. As key is being used the id of the block and the id of the proof, which is its index in the list, so both the value and the key are Strings. The rootHash is being stored in the memory of the Namenode and in the FSimage, so it can be retrieved during start of the HDFS.

The procedure of sending the data to the pipeline of Datanodes is still the same, nothing has changed. Datanodes know nothing about the proofs and the rootHash.

So far we have analyzed the procedure on the Client Side, during the write operation of the file. So the Namenode, have in storage the information about the rootHash and the proofs. That was the first part of the system regarding the Merkle Trees.

5.3.2 Block Report

A BlockReport happens every 6 hours by default, but we can also trigger a BlockReport with the command hdfs dfsadmin -triggerBlockReport ip:port (by default the port is 50020. The BlockReport is always initiated by the Datanode, since as we mentioned again Namenode never initiates communication with the Datanodes. We enhanced the BlockReport operation using Blockchain and the Merkle proofs mentioned above. The idea of the Block Report is to check the validity of the Blocks. The Namenode receives the Report and validates if the Blocks that are stored in a certain Datanode are not corrupted. The mechanism being used to validate the results are Merkle Proofs. Datanode is obligated to provide some Merkle Proof that the Namenode must validate and act depending on the result. The invalid blocks are being deleted from this Datanode. The procedure is being explained on the next paragraphs.

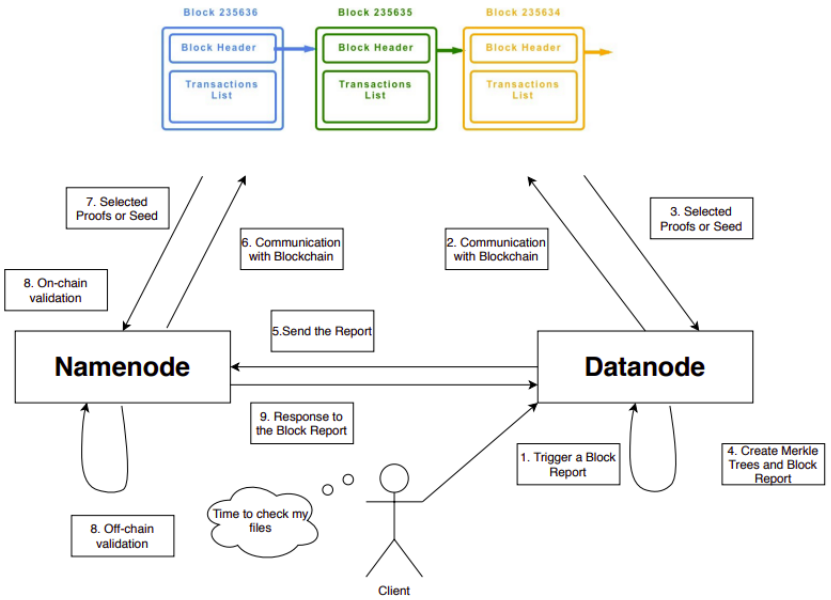


Figure 5.1: Architecture of Block Report

The main idea is that every time a BlockReport is being trigger the Datanode will be told to provide some Merkle proofs. For every block that the Datanode has must sent a given number of proofs, that

”someone” asked it. The first idea was the Namenode to be that ”someone” who determines the number and the specific proofs that the Datanode must give, but in case the Namenode is corrupted and collaborating with the Datanode this method can’t be applied. These nodes can cheat for their own benefit and no one will know about their trespass. For example, the Namenode may determine to ask for the same proofs every time, because it knows that the data of these proofs aren’t corrupted and let the Datanode corrupt all the other data. So, based on this assumption that every node can be malicious we made the decision to leave the determine process to a private Blockchain that is a trustworthy network, through the execute of smart contracts that we wrote, we will analyze the code and the function of the Smart Contracts below.

Now that we have talked about the determine process, let’s take a look at the procedure of sending and validating the Merkle Proofs. Think that the blockchain returns to the Datanode a list of numbers (we will analyze the procedure below, but let’s take it for granted that the Blockchain returns a list of numbers to the Datanode). The Datanode takes that list and determines which numbers correspond to which block (that depends on the Merkle Proof procedure that we decided to use) . So for example we have the list [0,1,3,1,2,6,3], the Datanode now knows that for the first Block must provide the proofs 0,1,3 for the second Block 1,2,6 and for the third Block proof number 3. So for every block that the Datanode must sent proof it creates its Merkle Tree, as long as Datanodes have in disk the real data of the file. The creation of the Merkle Tree is being explained above. Since the proofs are being asked for almost every block stored in the Datanode we used threads to parallelize the procedure to improve the performance, every thread available creates a Merkle Tree of a different block. The numbers correspond to the index of the proofs of the created Merkle Tree (the proofs are the leaves of the tree). Since the Tree is created and the proofs that the Datanode must give were identified it is time to provide the information needed for thee validation process. The procedure is explained in the 3.2.1.4 Merkle Proofs section. Once the path is created, a MerkleRequest object is being created. This object is part of the BlockReport and contains, the paths we created, a list that specifies if the corresponding hash in the list of the paths is the right or the left child, the ids of all blocks that participate in the MerkleRequest, the data (the 10000 bytes that the leaf was created, this makes the procedure even more secure since the Datanode proofs that it also contains the data).

Subsequently, the BlockReport is being sent to the Namenode, so it must validate the results. As soon as Namenode receives the BlockReport, it connects to the Blockchain and retrieves the results of the list of numbers the Blockchain returned to the Datanode earlier. This is achieved through events that the Smart Contract emits. Events are dispatched signals the smart contracts can fire, that includes information of the execution or data of a Smart Contract. So the Namenode using a filter of the time the transaction happened retrieves the specific list of numbers. There are two ways of validating the results. Since the Namenode can’t be trusted we implemented an On-chain validation and an off-chain validation too. The on-chain validation is more secure since the result of the smart contract can always be trusted, instead of the one which is provided by the local validation on the Namenode. The Namenode can corrupt the results and provide invalid information to the system. On the other hand the execution of the smart contract is stored on the chain and anyone can read it and confirm the Namenode’s action anytime. The algorithm is the same for the two cases, so will analyze in the next section the On-chain validation, so let’s focus on the Off-chain. Let’s take a look at the main functions that executes the validation process:

```
for every block:
   for every proof:
      root = findtheroot(path, parents, proof);
      if(root != storedRoot and hash(data) == SoredProofInDatabase):
         invalid.add(block);
```

Figure 5.2: First Step of Validation

For each block all the required proofs are checked. The function findtheroot is being called, which finds the root of the tree from the paths sent by Datanode, as explained in the Merkle Trees section, the function is presented below:

```
function findtheroot(path, parents, success):
   while(paths not empty):
      if(parents(0) == 1):
         success = hash(success, paths(0));
      else if(parents(0) == 0):
         success = hash(paths(0), success));
      remove(paths(0), parents(0));
return success;
```

Figure 5.3: Find the Root

The above function returns the root found by this process, we observe that the parent list contains either the number 1 or 0 to know in which order the two sequences should be hashed. Finally, the first function we mentioned checks if the root found by this process is the same as the one stored in Namenode memory. It also checks if the hash of the data stored by Datanode (the data from which the proof is obtained) is the same as the hash stored in the Rockdb database. If they are not the same, the block is added to the invalid blocks and Datanode is asked to delete it.. This is the case which the Namenode can be trusted. We can understand that a malicious Namenode can change the results since everything executes on its CPU. So it is essential that we execute the validation on-chain if we want our system to be almost 100% secure.

Blockchain

Let's take a look to the connection with the private Blockchain we have set up. In order to participate in a Blockchain transaction you must own a wallet. So every Datanode and Namenode must have a wallet (an address). A wallet in the ethereum is a .json file that the program of the Namenode and Datanode reads when the initialization procedure starts. The path of the wallet (.json file) is specified in the walletdn.xml (for the Datanodes) and in the walletnn.xml (for the Namenode). The connection

with the private Blockchain occurs in the ConnectiontoEthereum.java class, using the web3j library. Moreover, if the path of the wallet is empty the program creates a wallet for each node.

We set up the private blockchain using the geth which is a command-line interface (CLI) tool that communicates with the Ethereum network and acts as the link between your computer and the rest of the Ethereum nodes. To run a private network, you need to provide geth with some basic information required to create the initial block. Every blockchain starts with a Genesis Block, the very first block in the chain. To create our private blockchain, we will create a genesis block with a custom genesis file. Then, ask Geth to use that genesis file to create our own genesis block. The file of the genesis block looks like this:

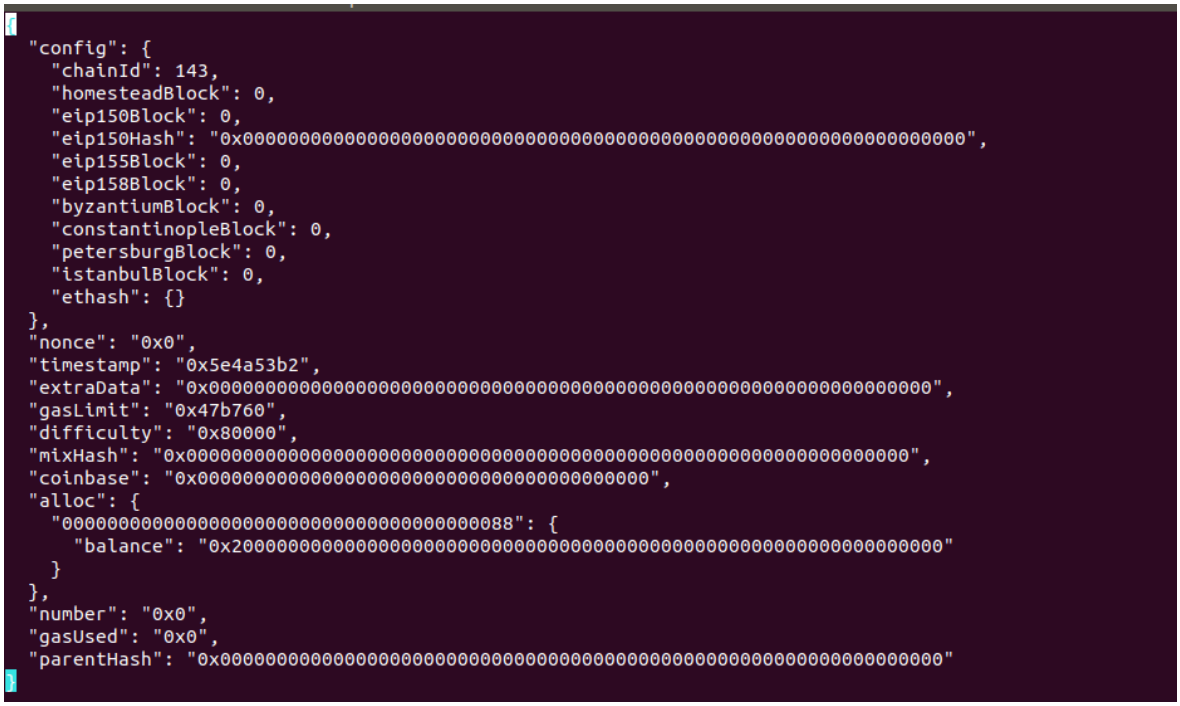


Figure 5.4: Genesis Block

Explanation on the config file;

To create the genesis block of the private blockchain, developers must set up a genesis file using a text editor, which is a '.json' file with the required configuration parameters described below: chainId: A unique identifier of the new private blockchain;

homesteadBlock: Homestead is the first production release of Ethereum and since the developers are already using this version the value of this parameter can be left as '0'.

Eip155Block/eip158Block: A few Ethereum Improvement Proposals (EIPs) were implemented to release Homestead, and required a hard fork. In a private blockchain development project such as this one hard forks aren't needed, hence the parameter value should be left as '0'.

Difficulty: In the Bitcoin network, miners solve an asymmetric cryptographic puzzle to mine new blocks. Over time the puzzle becomes easier, resulting in it eventually taking less than 10 minutes for each new block generation. Hence, the community updates the puzzle every 14 days and makes it more difficult, thus requiring even more computing power to handle the POW algorithm. The 'difficulty' parameter controls the complexity of the cryptographic puzzle. This parameter is also used in the Ethereum blockchain as well. Developers should assign a low value (between 0-10,000) to this parameter for this project thus enabling quicker mining.

gasLimit: If a transaction exceeds this limit then it won't execute in the Ethereum blockchain, hence, the developers should set a high value to this parameter for this project.

alloc: This parameter allows allocation of Ethers to a specific address, and isn't used for this project since the developers will mine dummy Ethers.

coinbase: also known as etherbase is the default primary account. You will get a warning if this value is not set but that can be safely ignored for now, you can set it later. difficulty: This determines how difficult it is to mine a block. For a private blockchain it is better to set it to a low number to ensure blocks are mined quickly which utlimately translates to faster transactions. extraData: Block extra data, it defaults to the client version if not provided.

gasLimit: This dictates the maximum amount of gas that can be used in each block. The higher the value the more transactions can be squeezed into a block.

mixHash: Not relevant to a new private network, set to 0

parentHash: Not relevant to a new private network, set to 0

Geth requires two main parameters, a folder to store the chain data (i.e. local database) and an initialisation file. We run the command geth –datadir ./chain init genesis.json to initialize our private Blockchain. And then we run the command geth –': geth –allow-insecure-unlock –port 3000 –networkid 143 –targetgaslimit '90000000000' –nodiscover –datadir=./chain –maxpeers=0 –rpc –rpcport 8543 –rpcaddr 147.102.4.137 –rpccorsdomain https://remix.ethereum.org –rpcapi "eth, net, web3, personal, miner" to make the Blockchain start. Subsequently, we created the wallets and put them at the specified paths. We set the coinbase (the Namenode's wallet).

We deployed the Smart Contracts in the private Blockchain using truffle and took the addresses of the smart contracts in order to communicate with them through the java code using web3j.

Smart Contracts

In this section we analyze the smart contracts we run on the Blockchain. There are two reasons to contact the Blockchain. The first one is to take the proofs that the Datanode must give. This procedure can either return a list of numbers or a seed is being used to generate deterministic pseudo-random sequence. The second one is the on-chain validation of the results on Namenode side.

Smart contracts are simply computer programs. The word contract has no legal meaning in this context. Once the smart contract is deployed, the code of it cannot change. Unlike with traditional software, the only way to modify a new contract is to deploy a new one. The outcome of the execution of a smart contract is the same for everyone who runs it. The EVM runs as a local instance on every Ethereum node, but because all instances of the EVM operate on the same initial state and produce the same final state, the system operates as a single "world computer". As we mentioned above, there are 2 reasons to communicate with the Blockchain. The first one is for determination of the proofs and the second is for the validation. At first let's explain how the determination of the proofs occurs. The determination of the proofs depends on the validation process we decided to use. We can choose over three options. The first one is one proof per block, thee second one is k proofs per block and the third one is n proofs from all the blocks. Lets examine the first one. We go through all blocks and make a list with numbers that contains how many potential proofs each block has. So we contact the Ethereum using a Java Wrapper Class which was generated directly from a smart contract to interact with a smart contract in Java. We used the command web3j solidity generate -a=<abiFile> -b=<binFile> -o=<destinationFileDir> -p=<packageName> which results the creation of the Wrapper. The abiFile and binFile is generated during the compilation of the Smart Contract written in solidity. Below we see the Smart Contract written for the three options of determination mentioned above:

```
contract Numbers:

  function number(data, time):
    for(i=0 until i<data.length):
      result.add(hash(data(i), i, time) mod data(i));
    emit event NumberIssue(time, result, senderAddress);

  function numberK(data, time, k):
    for(i=0 until i<data.length):
      for(j=0; until j<k):
        result.add(hash(data(i), time, j, i, k) mod data(i));
    emit event NumberIssue(time, result, senderAddress);

  function kproofs(data, time, k):
    total = 0;
    for(i=0 until i<data.length):
      total  = total + data(i);
    step = total div k;
    begin = total mod k;
    for(i=begin until i<total; i=i+step):
      result.add(hash(total, time, i) mode total);
    emit event NumberIssue(time, result, senderAddress);
```

Figure 5.5: Smart Contract

The first function number corresponds to the first option. It takes as parameter a list of numbers of the proofs exist in every block and the time of attending to communicate with Ethereum. So we perform a hashing between the data, the index of the data in the list and the time. The number found is being normalized through the modulo operation with the data. We want the result to be between zero and data[i]. Think that the maximum index of the proof that can be asked in each block is the number data[i]. So we want the result to be between these values.

We observe that the second option of k proofs on each block is the same with the addition of an extra for-procedure since we want to take k proofs on each block. So the hashing is now between data, the index of the data, the number of the proof (0-k) and time.

The third option is a little bit more complicated. The idea is to get n random proofs from all the blocks. So at first we summarize all the values of the numbers of the proofs that exist in every block. As a result we take the number of all existing -potentially asked- proofs in the Datanode. Since we only want n of them we normalize the for procedure with the use of division and modulo operations. Now the hashing is between the total number of the proofs, the time and the index of the asked proof (0-n) modulo the total value in order to normalize the value of the asked proof between zero and total number of proofs. The result is being returned to the Datanode which determines which proof corresponds to which block. Then the procedure of the BlockReport is the one mentioned above.

Another way of determining the proofs that the Datanode must give is through a seed. A seed is a random number that is the start point when a computer generates a random number sequence. The idea is that the Blockchain instead of sending a whole list of numbers and consume time for transferring big amount of data is to provide a number called seed. The Datanode takes the seed and through a deterministic process it creates a sequence of numbers in range [0,total], total is the total amounts of proofs that a Datanode possess. Below there is the Smart Contract we used for creating the seed and the implementation on java for creating the sequence of numbers.

```
contract Seed:

  function seed(time):
     s = hash(time) mod 2147483647
     emit event Seeds(time, s);
```

Figure 5.6: Seed

```
Random rand = new Random(seed);
for(i=0 until i<NumberofProofs):
   proofs.add(rand.nextNumber mod total);
```

Figure 5.7: Creating the Sequence

The second reason that the program contacts the Blockchain is the on-chain validation of the proofs given. The Namenode contacts the Blockchain in order to execute the Smart Contract that corresponds to the On-chain Validation. The procedure is the same as the off-chain validation. The execution of the smart contract returns a Boolean List containing the results of the validation. The Namenode parse the results and sends to the Datanode which blocks are invalid. The result of the validation is now on-chain so anyone can retrieve and validate the actions of the Namenode, so every attempt to corrupt the results can be verified.

Chapter 6

Experiments and Conclusions

The experimental procedure took place on the laboratories server. We used the clones to set up the system. We used one Namenode, three Datanodes and one machine to make the private Blockchain run.

6.1 HDFS

The first set of experiments concern the HDFS (the classic Hadoop without the extensions we added). The experimental procedure includes the timing of the put operation during the upload of different size of files with replication factor equals to two. We performed five times each experiment and we calculated the average value. The results are presented below:

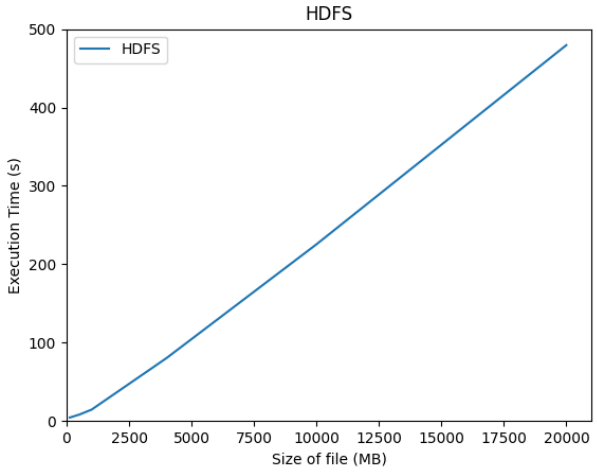


Figure 6.1: Experiments of HDFS

6.2 Put Operation

Then, measurements were made on the Hadoop-Ethereum that includes the Merkle Trees. As before, timing was done in the put command, measuring both the total time and the individual time it takes the client to build the Merkle Tree for each block, as well as the time it takes to send the Merkle Tree data to Namenode (root hash and leaves). The measurement was performed 5 times, so for each block we took the average of 5 measurements and then we added the result we got for each block for the sizes we mentioned above, to see how much time was consumed in total in the specific sizes. Different Merkle Chunk sizes were used. Also, the command put were given from the Namenode, later we are going to examine the performance if the command is being given from a different machine The results

are presented below:

Για Merkle Chunk 512 bytes:

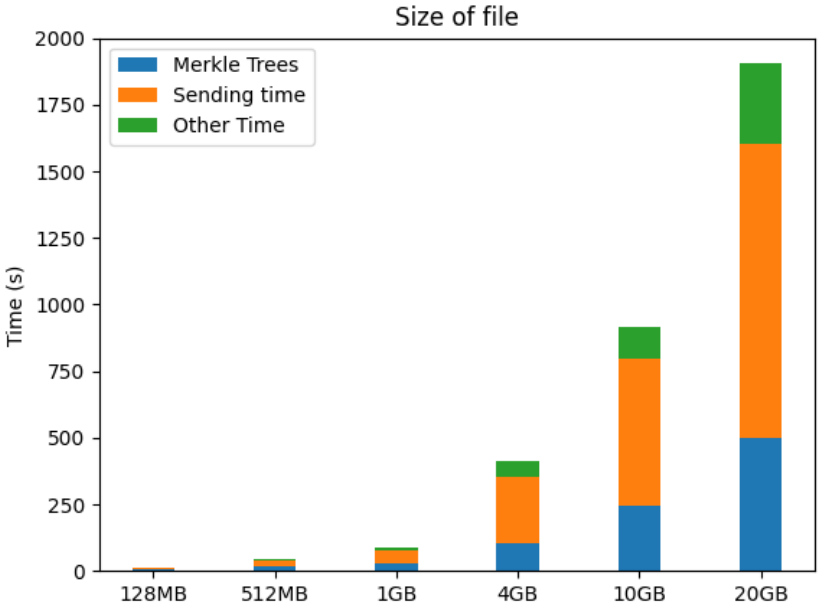


Figure 6.2: Experiments for Merkle Chunk 512 bytes

Για Merkle Chunk 1024 bytes:

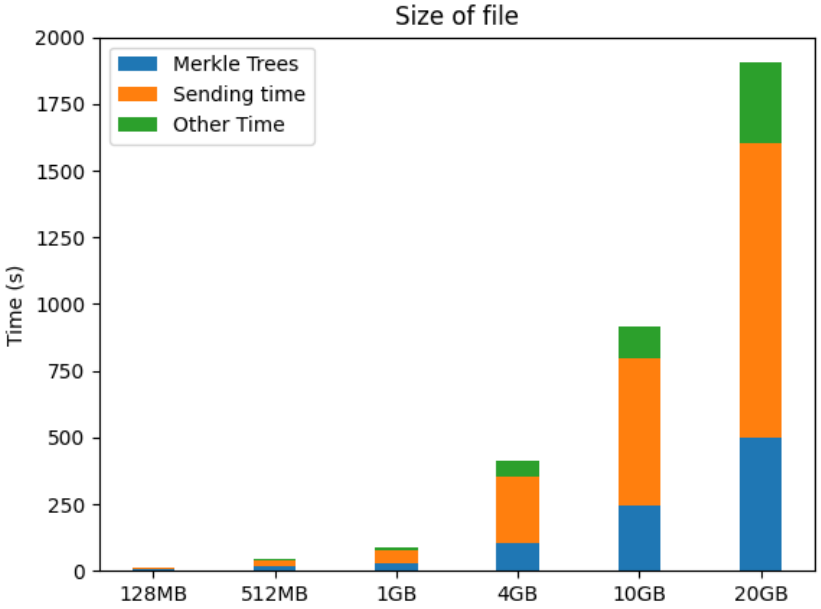


Figure 6.3: Experiments for Merkle Chunk 1024 bytes

Για Merkle Chunk 10000 bytes:

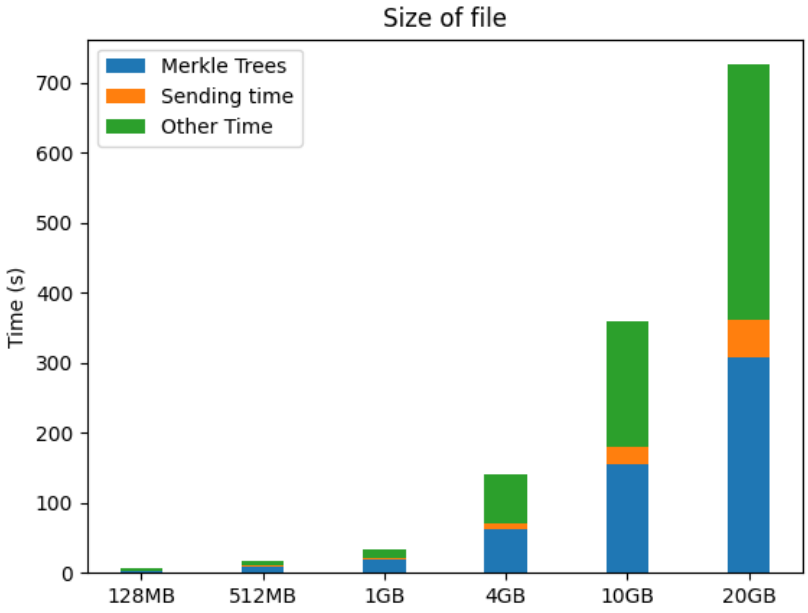


Figure 6.4: Experiments for Merkle Chunk 10000 bytes

Για Merkle Chunk 100000 bytes:

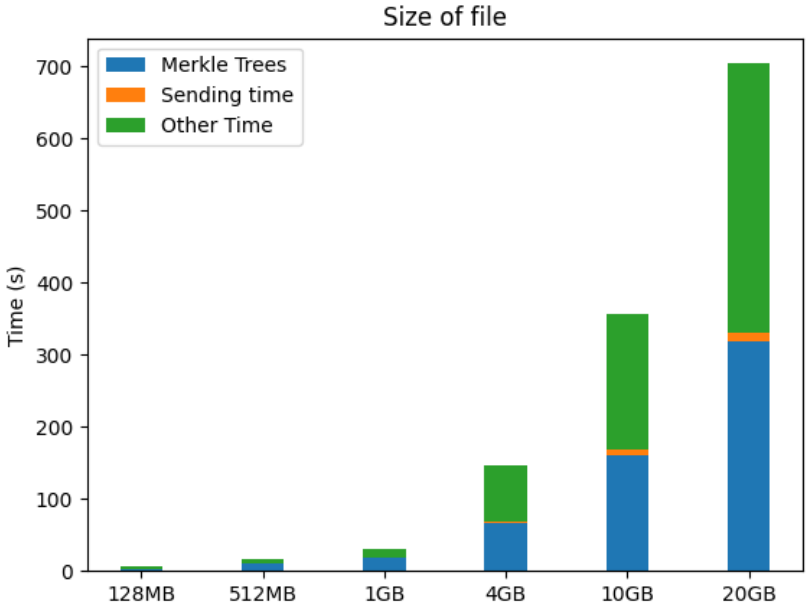


Figure 6.5: Experiments for Merkle Chunk 100000 bytes

Για Merkle Chunk 1000000 bytes:

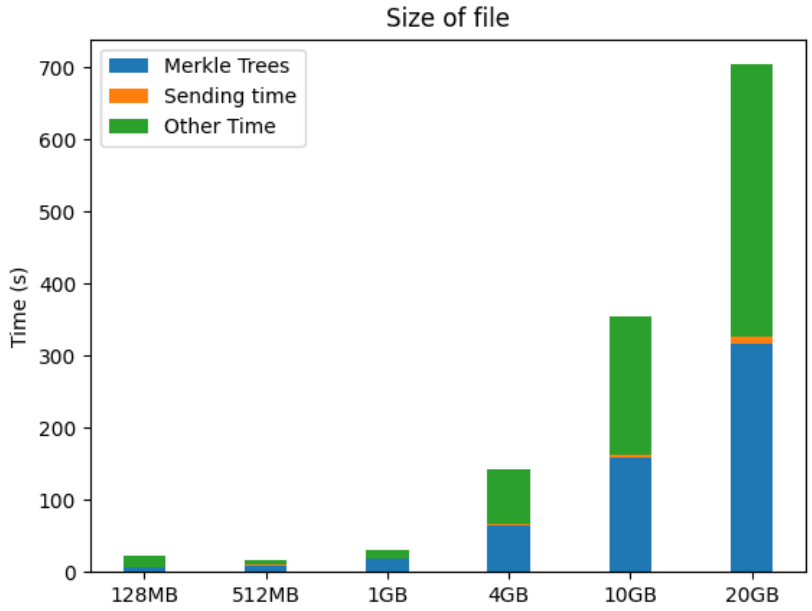


Figure 6.6: Experiments for Merkle Chunk 1000000 bytes

Below we present a Figure containing all the Experiments:

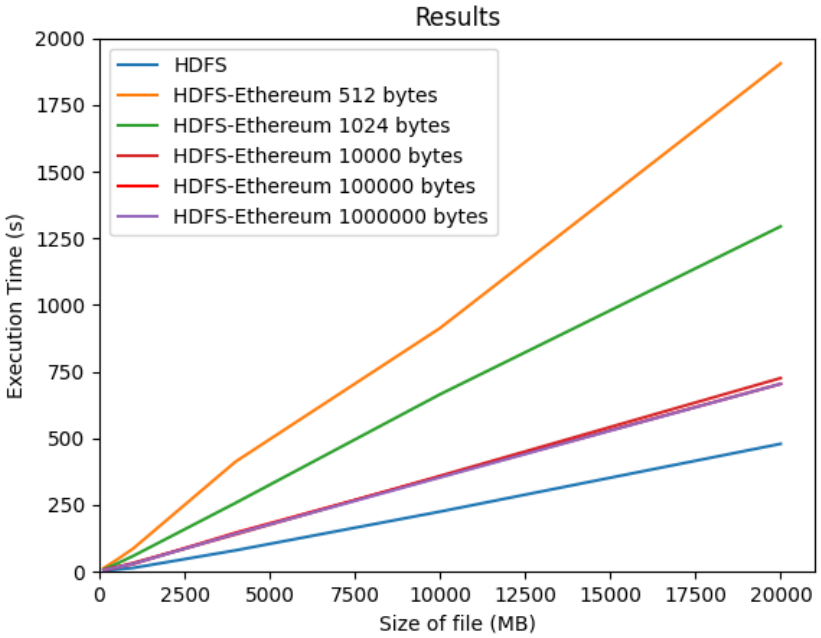


Figure 6.7: Total Time of All Experiments

We observe that after 10.000 Merkle Chunk size and above the time that takes for the file to upload is almost the same. So it is more secure to have a bigger Merkle Tree, so we decided that the 10.000 bytes are the most efficient size to keep in balance both performance and security of the system.

The above measurements were performed considering the namenode itself as a client. Using

the 10.000 bytes merkle chunk, which was considered the most efficient, the measurements were performed again, considering a different node as the client. The results are presented below:

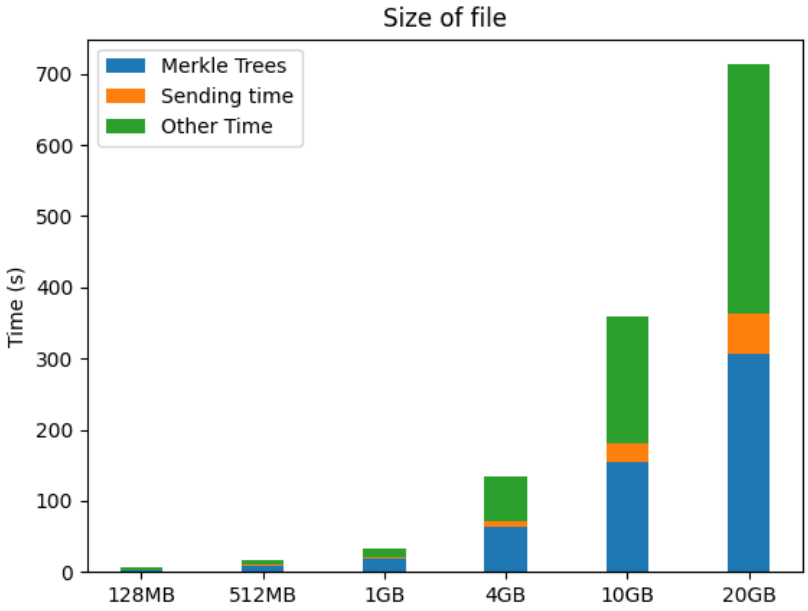


Figure 6.8: Different Client

6.3 Blockchain

The Blockchain measurements were then performed. More specifically, we measured the time Namenode to retrieve the information from the Blockchain and validate the BlockReport, depending on the number of proofs requested by the Datanode. We also measured the time it takes for a Datanode to Build the BlockReport from scratch, so including the communication with the Blockchain. The determination process for the proofs that the Datanode must give is being examined with the use of seed and the list of numbers as well. Validation can be done either locally or on-chain. Both cases are considered below:

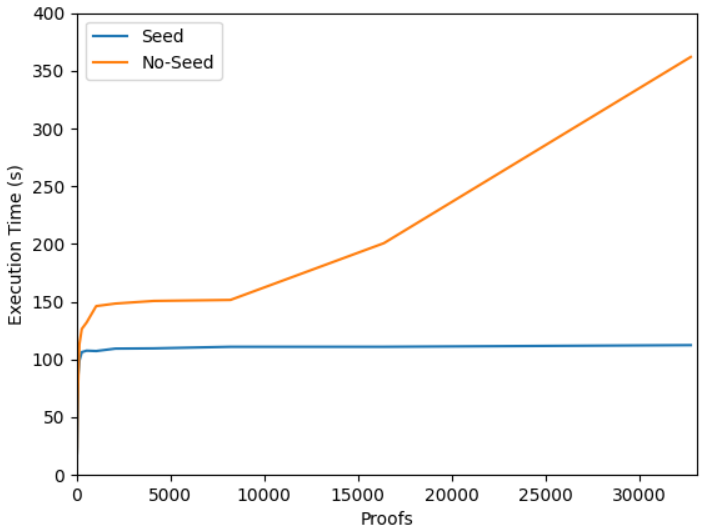


Figure 6.9: Datanode Seed vs no-Seed

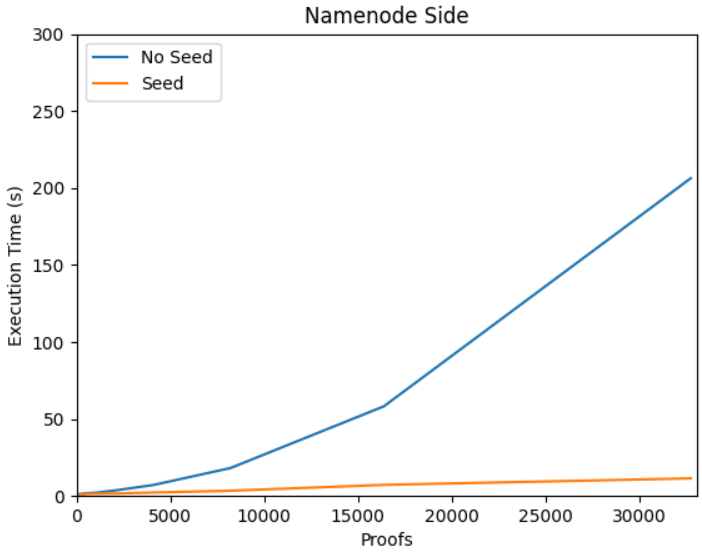


Figure 6.10: Namenode Seed vs no-Seed

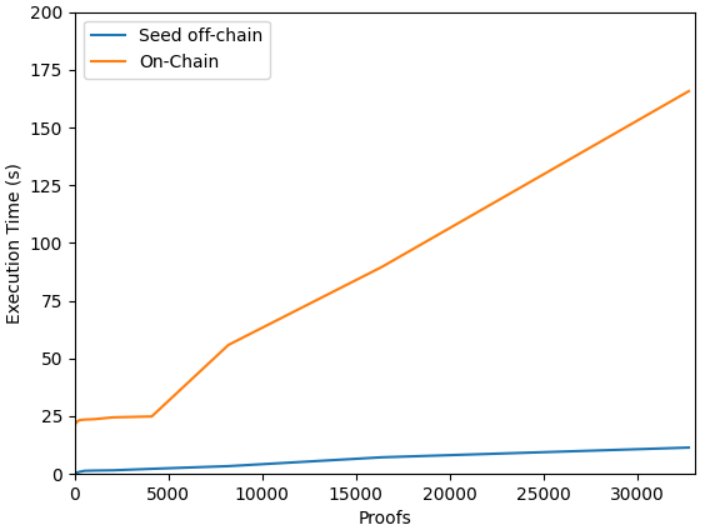


Figure 6.11: Namenode On-chain vs Off-chain

From figure 6.9 we observe a significant improvement in time with the use of Seed. The transfer and recording of large amount of data is a suppressive factor for the operation of Blockchain. With the use of Seed, however, where the Smart Contract returns only one number, the improvement is significant. Also the time remains relatively constant from a number of proofs and above. As it creates the same number of Merkle Trees. The Datanode in which the experiments were performed had 49 blocks. We notice that from the number of proofs 256 and above Datanode created 49 trees which correspond to all its blocks. So it makes sense for time to be similar, as the processing volume is the creation of Merkle Trees from files.

From Figure 6.10 we observe that Namenode has the same behavior with respect to Seed. With the use of Seed the time is significantly improved, as there is no waiting in the transaction with Blockchain for the transfer of a large amount of data. The Smart Contract returns a number and Namenode undertakes to create the sequence of numbers.

From figure 6.11 we observe that On-chain validation is more time consuming than both of the above methods, as expected. Blockchain takes about 20 seconds to mine a block. The number of proofs we can send in each transaction for validation is 128. So we observe that until the number of 5.000 proofs the time remains constant. The time increases depending on the batches we send, since each block contains up to 32 transactions of 128 proofs each.

The seed method is obviously better for determining the proofs that the Datanode must give, as it does not create any defect and greatly improves the performance of the program. In terms of validation, On-chain is the only way to be sure of the outcome of the process, but it is quite time consuming and does not help the performance of our program at all. Off-chain, on the other hand, offers quick validation, but creates security vulnerabilities, as Namenode is not controlled from anywhere, so it must be considered reliable. Which method we decide to use is based on the platform we want to use this application. In case Namenode can be considered reliable it is clearly better to use Off-chain validation. Otherwise, it may be better to use a combination of methods. That is, some proofs to be validated 0ff-chain and some On-chain, a hybrid scheme that will offer us both security and a better performance.

Bibliography

[1] Andreas M. Antonopoulos and Gavin Wood. *Mastering Ethereum: building smart contracts and DApp*. 2018.

[2] Blockchain Applications. https://blockgeeks.com/guides/blockchain-applications/.

[3] Blockchain. https://www.upgrad.com/blog/why-blockchain-is-important/.

[4] Cloudera Blog. https://blog.cloudera.com/understanding-hdfs-recovery-processes-part-1/.

[5] Ederov Boris. Merkle tree traversal techniques. 04 2007.

[6] Elliptic Curve. https://fangpenlin.com/posts/2019/10/07/elliptic-curve-cryptography-explained/.

[7] Katerina Doka, Tasos Bakogiannis, Ioannis Mytilinis, and Georgios Goumas. *CloudAgora: Democratizing the Cloud*, pages 142–156. 06 2019.

[8] Wikipedia for Digital signatures. https://en.wikipedia.org/wiki/digital_signature.

[9] Geek for Geeks. https://www.geeksforgeeks.org/what-is-dfsdistributed-file-system/.

[10] Wikipedia Crypto Hash Functions. https://en.wikipedia.org/wiki/cryptographic_hash_function.

[11] Apache Hadoop. https://hadoop.apache.org.

[12] Apache Hadoop HDFS. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/hdfsdesign.html.

[13] Investopedia. https://www.investopedia.com/terms/b/blockchain.aspwhat-is-blockchain.

[14] Iveta Kremenova and Milan Gajdos. Decentralized networks: The future internet. *Mobile Networks and Applications*, 24, 01 2019.

[15] Medium. https://medium.com/@zhaohuabing/hash-pointers-and-data-structures-f85d5fe91659.

[16] Ralph Merkle. A digital signature based on a conventional encryption function. *LNCS*, 293:369–378, 08 1987.

[17] Muqaddas Naz, Nadeem Javaid, and Sohail Iqbal. *Research Based Data Rights Management Using Blockchain Over Ethereum Network*. PhD thesis, 09 2019.

[18] Merkle proofs. https://medium.com/crypto-0-nite/merkle-proofs-explained-6dd429623dc5.

[19] John Taylor Rowan, Garnier and. *Discrete Mathematics: Proofs, Structures and Applications*. 2009.

[20] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Rob Chansler. The hadoop distributed file system. *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on: 2010*, 26, 05 2010.

[21] DIstributed Systems. https://www.freecodecamp.org/news/a-thorough-introduction-to-distributed-systems-3b91562c9b3c/.

[22] Michael Szydlo. Merkle tree traversal in log space and time. volume 3027, pages 541–554, 05 2004.

[23] Hawaii University. http://itm-vm.shidler.hawaii.edu/hdfs.

[24] Maarten van Steen and Andrew Tanenbaum. A brief introduction to distributed systems. *Computing*, 08 2016.

[25] T White. *Hadoop: The Definitive Guide*. 01 2010.

[26] Wikipedia. https://en.wikipedia.org/wiki/distributed$_{f}ile_{s}ystem_{(}microsoft)$.

[27] K. Ziegler. A distributed information system study. *IBM Systems Journal*, 18(3):374–401, 1979.