Chapter 11
Dependability and Security

with Ralph Holz, Vincent Gramoli, and Alex Ponomarev

In this chapter, we discuss dependability and security aspects of blockchain-based applications and analyse how the different properties of dependability and security relate to these applications. As is the case throughout the book, our viewpoint for the discussion in this chapter is that of a system architect or developer using blockchain as a component. We thus analyse how blockchains impact the dependability and security of systems built upon them, in part with studies using observations from the mainstream proof-of-work blockchains Ethereum and Bitcoin. As such, we are not going into the details of cryptography and security infrastructure of blockchain platforms.

Dependability and security are tightly interlinked. According to the widely accepted taxonomy of Avizienis et al. (2014), dependability and security are comprised of six attributes as shown in Fig. 11.1. The first five sections of this chapter give an overview of the influence on dependability and security attributes of blockchain as a component within a multiparty system.

We then focus on the availability of functions that such systems need, in particular transaction inclusion, and how they may be adversely impacted by a number of factors. When viewing blockchain as a component for data storage, communication, or code execution, whether a transaction is included or not can largely be equated to write/send availability.

Finally, in Section 11.7, we discuss issues around aborting and retrying transactions—a functionality that is not provided by blockchain client software today.

X. Xu et al., *Architecture for Blockchain Applications*,
https://doi.org/10.1007/978-3-030-03035-3_11

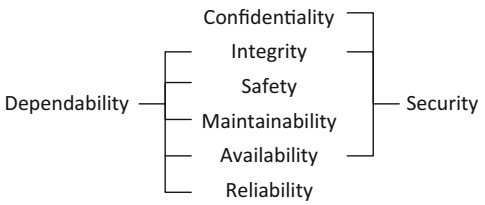

Fig. 11.1 Attributes of security and dependability. © 2004 IEEE. Reprinted, with permission, from Avizienis et al. (2014)

11.1 Confidentiality

Confidentiality means that unauthorized disclosure of information does not take place. This is usually harder to establish in blockchain-based systems, because the default is that information is visible for everyone in the network. Information can be encrypted: asymmetrically with a particular party's public key, so that only this party can decrypt it, or symmetrically with a shared secret key, so that the group of parties with access to the secret key can decrypt it. The latter case requires a secure means of exchanging the secret key, typically off-chain. However, once information needs to be processed by smart contract methods, this information needs to be decrypted. This is because smart contract code runs on all nodes of the network, and thus any of them needs to be able to process the input data. The ability for anyone to execute smart contracts is required to achieve consensus on the outcomes of smart contract execution. Embedding keys within a smart contract would reveal the key to all participants.

As discussed in the supply chain use case in Section 4.1, commercially sensitive data can be at risk if it is shared on a blockchain, even if pseudonyms are used and even if encryption is used. Private and permissioned blockchains can provide read access controls, but this will not provide commercial confidentiality between competitors using the same blockchain.

There are interesting technologies on the horizon, which could alleviate some of these pain points. For instance, zero-knowledge proof methods like *zk-SNARKs* can be used to hide the contents of a transaction, while still allowing independent validation of the integrity of that transaction. Current implementations are limited to hiding simple transfers of cryptocurrency, but in the future the same could be achieved for more sophisticated transactions. As for computation on encrypted data, that is the goal of techniques like *homomorphic encryption* and *confidential computing*. However, such approaches have not been utilized for smart contracts as yet, in part due to their significantly increased computational requirements over regular computation. Alternatively, authorized 'witnesses' could have special access to the data. These witnesses could be certifying agencies or consumer group advocates. The data would be encrypted using the witness' public key, so that only the witness can decrypt it. The witness can then pass on the provenance information to interested parties, but not share information that is commercial in confidence. How the data is to be encrypted and stored would be part of the smart contracts created for various supply chain events and, as such, can be customized for different scenarios and supply chains.

11.2 Integrity

Integrity is the absence of improper (invalid or unauthorized) system alterations and is a key attribute for blockchains. Once a transaction is included in a blockchain and committed with sufficiently many confirmation blocks, it becomes part of the effectively immutable ledger and cannot be altered. This also applies to smart contracts: their bytecode is deployed in a transaction and thus is subject to the same integrity guarantees; and invocation of smart contracts happens through transactions as well. The key integrity property of Bitcoin is that addresses cannot spend money they do not have. Ethereum's integrity property is more complicated, because it requires the correct operation of a Turing complete smart contract programming language. However, for client applications, Ethereum provides significant power by allowing user-defined integrity conditions to be implemented as checked preconditions and defined behaviours in smart contracts.

Blockchain emerged to support a cryptocurrency, and so it is unsurprising that integrity is a key dependability attribute, because integrity is the key dependability attribute for commercial computer security. The seminal work on this topic is the Clark–Wilson security policy model, and blockchains are broadly consistent with its requirements. Smart contracts can implement Clark–Wilson's transformation procedures to generate and update internal data or other smart contracts that realize Clark–Wilson's constrained data items. Blockchains natively create the log required by Clark–Wilson for reconstructing operations. Finally, blockchains use a kind of separation of duty through the replicated validation performed by all mining nodes.

Ethereum smart contracts are written in a Turing-complete programming language. This makes it more difficult to verify that the smart contracts correctly implement required integrity properties. Formal verification techniques can be used, but these can be costly and time-consuming in practice. A lighter-weight approach is to use a smart contract language with strong typing mechanisms, which can help programmers support integrity. The Pact language on the Kadena blockchain[1] is an example of that approach. Some blockchains, such as Kadena and Corda,[2] avoid the use of Turing-complete smart contract languages for this reason, and instead use less-expressive domain-specific languages that can be automatically checked.

High integrity and non-repudiation are not always ideal. For example, sometimes historical data must be deleted or changed. If a vexatious or improper registry entry has been created, a court may order the registrar to change the registry to remove that entry, 'as if it had never been created'. This is not technically possible on many blockchain platforms. Similarly, this may create problems for blockchains that have been 'poisoned' by illegal content. Some blockchains have been proposed to deal with this challenge, but there is not yet widespread acceptance and adoption of good solutions.

[1] http://kadena.io/.

[2] https://www.corda.net/.

11.3 Safety

As defined by Avizienis et al. (2014), safety means that using a system does not lead to catastrophic consequences on the users and the environment. The use of blockchain technology does not directly pose specific risks in this regard, when compared to other components in distributed systems. There may be economic and environmental risks from investments in ineffective blockchain mining strategies. By using non-mining nodes, private or permissioned chains, and/or alternative consensus mechanisms, these risks can be mitigated. If a blockchain is used as a component in a safety-critical application, then failures of integrity, confidentiality, availability, or other dependability attributes may have consequences for safety. However, this is a prevalent issue of safety-critical system engineering. A noteworthy difference exists when the cryptocurrency or token features of a public blockchain are used, in which case an organization or user is exposed to the monetary risk of loss or devaluation of the cryptocurrency and tokens. If this risk can bankrupt the organization or users, it may lead to situations that could be seen as catastrophic. With respect to cryptocurrency, a difference to regular internet-related flow of monetary assets lies in the fact that there is no additional safety net. No banks will stop attacks on your Bitcoin wallet or reimburse your losses. In most cases of theft, lost crypto-coins or tokens remain unrecoverable.

An alternative, informal definition of safety by Lamport (1977) states that some 'bad thing' does not happen during execution. Alpern and Schneider (1985) later formalized this definition with regard to discrete execution states of programs but did not formalize what a 'bad thing' might be due to the inherently informal nature of this concept. Examples of safety properties mentioned in the above sources are mutual exclusion of concurrent processes, deadlock freedom, partial correctness, and first-come-first-served execution.

This perspective is indeed interesting when considering blockchain. When considering a public blockchain network execution itself as the program, discrete states are almost meaningless: the states of different nodes around the world are only very loosely synchronized, and substantial differences between, say, the current transaction pools of set of nodes can be expected. Considering the committed blocks in a blockchain (e.g. the current Ethereum blockchain minus the most recent 11 blocks), discrete execution states become a valid model. Concerning the above-quoted examples, concurrent processes and *mutual exclusion* are a non-issue (since the execution has been sequentialized).

Deadlocks on the application level can exist as in any other program, be it on the smart contract level or off-chain programs. It might be easier to avoid deadlocks in blockchain-based applications, since smart contracts can be used as a neutral mediator, which handles all resources (e.g. cryptocurrency in exchange for tokens), instead of distributed processes responsible for different resource types.

Though not mentioned in the early literature, *livelocks* or infinite loops in a smart contract would, in the case of Ethereum, be resolved by the platform itself: each invocation has a limited amount of gas available. If the execution does not terminate before the gas has been consumed, it is aborted.

Partial correctness on the application level is not impacted by blockchain. However, correctness of the execution when computing a new block is higher: in many public blockchains, all full nodes verify each newly announced block by checking digital signatures and hashes and executing all transactions. If the results—e.g. of a smart contract invocation check—are inconsistent, the new proposed block is discarded.

If *first-come-first-served* is required, typically that requires special consideration on blockchain. In Bitcoin, strict ordering of transactions can be established by consuming an output of one transaction as input of another transaction—the second is only valid once the first has been included, although both transactions can be included in the same block. Similarly, Ethereum transaction nonces can be used to ensure ordering of transactions, but this feature is only available for transactions originating from a single account. A smart contract can ensure ordering to a degree, e.g. if the order can be prescribed. For scenarios where neither of these options suffice, e.g. open bidding processes, off-chain mechanisms might be required to ensure fair processing. Generally, the inclusion of a given transaction is not guaranteed by blockchain protocols, let alone in any particular order. This issue of availability from the viewpoint of an application will be discussed in depth in Section 11.6.

11.4 Maintainability

Maintainability refers to a system's amenability to undergo modifications and repairs. In blockchain-based systems that use smart contracts, this is harder to implement for the smart contracts than in regular distributed systems. This is because smart contracts comprise code that regulates the interactions between mutually untrusting parties; trust is derived from the fact that the code cannot be changed easily. Consider an example where an organization has established trust in the code of a particular contract and verified that it implements the agreed rules for handling cryptocurrency. If others can change the code without that organization's knowledge or consent, any trust in the code would be void. Although the code of an Ethereum smart contract cannot be changed, the current state of variables within that smart contract can be updated by invoking its methods. In particular, these variables may refer to other smart contracts. This mechanism provides a kind of indirection that allows the dynamic updating of smart contract code, through mechanisms like the Contract Registry Pattern (Section 7.4.1). However, support for this kind of updating must be specifically provisioned ahead of time.

Finally, changes may be made to a blockchain-based system not by changing the data stored on a blockchain but instead by changing the interpretation of data on the

blockchain. As an extreme example, a client application might choose to not honour all data previously written to the blockchain under some previously acknowledged addresses. Instead, the client could in principle re-create all required data on the blockchain under some new address. A distinctive benefit of blockchain-based systems is that there is no single party with control of the system. However, this inherently creates challenges for governance: the management of the evolution of blockchain-based systems. Changes may be made to correct defects, add features, or migrate to new IT contexts. However, in a multiparty system with no single owner, managing these changes is more like diplomacy than traditional risk management or conventional product management. Lessons may be drawn from governance in open-source software, which face similar development challenges. However, the governance of a blockchain is not just a software development problem—it is also a deployment and operations problem. For both public and private blockchain systems, key stakeholders include the users of the blockchain, software developers with moral or contractual authority over the code base, miners or processing nodes in the blockchain ecosystem, and government regulators in related industries. There are still lessons being learned about who the key stakeholders for blockchains are. For instance, the 2016 hard fork of Ethereum in response to the DAO controversy made it apparent in hindsight that digital currency exchange markets are a key stakeholder for public blockchain systems. (The market initially provided by the Poloniex exchange for trading the unforked 'Ethereum classic' digital currency has supported the ongoing operation of that blockchain, which might have otherwise failed to continue to be viable.)

It is unknown how to best perform governance for blockchains and blockchain-based systems. How should relevant stakeholders influence and manage changes to the software and the operational infrastructure for blockchains and blockchain-based system when there might be no central owner and where the blockchain platform might be serving many purposes for different stakeholder groups?

11.5 Availability and Reliability

According to Avizienis et al. (2014), availability is the readiness for correct service, whereas reliability is the continuity of correct service. More specifically in our context, availability concerns the users or dependent systems' ability to invoke functions of the system, whereas reliability refers to receiving consistently correct outcomes from those invocations.

For blockchains, there are scenarios in which the distinction between reliability and availability can be blurred as there is no globally specified time by which a transaction should complete. If a blockchain system never includes a transaction (perhaps because other connected nodes ostracize that transaction, address, or interface node), that will be both an availability and reliability failure of the blockchain system from the perspective of a client application. However, if a transaction is initially included in some block, that does not guarantee that block will

be recognized as being part of the blockchain in future. One could take the following view: first an application designer can specify a number of confirmation blocks by which they will regard a transaction to have been committed. If a fork happens that invalidates transactions thought to be committed, the system will have had a reliability failure, because a transaction thought to have been committed will have turned out not to be. Alternately, if a fork affects less than the specified number of confirmation blocks, the system may experience enough delay to have an availability failure.

The operation of public blockchains can involve hundreds or thousands of independent processing nodes. Each node holds a full replicated instance of the blockchain transaction history and can operate for users as a transaction interface to the blockchain network. Because of this massive redundancy, naively we might expect that a blockchain system has extremely high availability. We can assume that local components of a blockchain-based system are connected to a local full node on the blockchain network. Submitting a transaction to a blockchain network is done through the local full node, which broadcasts that transaction to all nodes it is connected to. The availability of a locally reachable full node is thus heavily reliant on the organization operating a blockchain-based system. The more complex question is: how certain can one be that the transaction is included in a block and committed, in a timely manner? We address this question in the next section in detail.

Transactions deploying smart contracts or invoking their methods add a further level of complexity. First, they are subject to more parameters, like current gas limit, that may impact their successful inclusion. Second, they utilize more complex functionality of the network and thus rely on the network sharing the same accepted norms about this functionality with the system. For instance, parts of the network may change to not accept certain commands present in compiled smart contracts. If the blockchain-based system is unaware of the change, it might attempt to use these commands, and its contracts might get rejected, or method calls might be terminated unexpectedly. Again, we discuss these issues in more detail in the next section.

Finally, we note that the well-known *CAP theorem* indicates that there is inevitably some trade-off between consistency, availability, and partition-tolerance for distributed databases. As described above, blockchain platforms sacrifice traditional notions of consistency, but strive for availability and partition-tolerance.

11.6 Variation in Blockchain Transaction Inclusion

Blockchains are distributed systems, and so the states of different parts of the system are inevitably different. Different nodes will hear about new pending transactions and new blocks at different times. There is also variation in how long it takes for the system to commit transactions in the ledger. For public blockchains like Bitcoin and Ethereum that use Nakamoto consensus, there is much greater variation in transaction inclusion time, which is exacerbated by the probabilistic nature of transaction inclusion.

In fact, there can be so much timing variation that it can impact core dependability attributes. If transactions takes *too* long to be included, they will violate *latency* or service availability requirements. Integrity can also be impacted if transaction reordering occurs because of the probabilistic nature of Nakamoto consensus. This section explores these issues in some detail, for both Bitcoin and Ethereum.

11.6.1 Variation in Bitcoin Transaction Commit Time

In this section, we explore the factors that impact Bitcoin commit time and show that reordering of transactions play an active role.

A peculiarity of Bitcoin is the way transactions are linked: they transfer currency from a number of source addresses to a number of destination addresses. Recall from Section 2.1 that transaction outputs become the inputs of new transactions. If the sum of the outputs is less than the sum of the inputs, this is interpreted as an additional output that pays a fee to the miner who mines the block containing this transaction. This acts as an incentive for miners. As a result, miners tend to optimize block creation by preferring transactions with higher fees. The transaction fee is often the only variable that client software asks Bitcoin users to choose consciously when creating a new transaction.

However, transactions can also experience delay due to other factors. An important one is that transactions must arrive (roughly) in order, for a node (and the network) to be able to process them fast. Incoming transactions are handled by the so-called mempool. If the referenced input transactions (the 'parents') are yet unknown, a miner will delay the inclusion of the new transaction—it is then a so-called 'orphan'. Miners may choose to keep orphans in the mempool while waiting for the parent transactions to arrive, but they may also expunge orphans after a timeout they choose. A second factor that could come into play, albeit one that only experienced users will set, is so-called locktimes: a transaction can contain a parameter declaring it invalid until the block with a certain sequence number has been mined. This makes it possible to set an 'execution date' for transactions.

Note that out-of-order arrival may be the result of a number of factors: the forwarding behaviour of a node depends on the implementation and is different even between versions of the 'official' Bitcoin Core client. It may naturally also depend on the load on miners (leading to low throughput as evidenced by an ongoing community discussion[3]). Transient connectivity issues and Internet routing constellations may also be at play.[4] Also note that transactions may be rejected by the mempool for certain reasons. We explain these below as we encounter them.

[3] https://en.bitcoin.it/wiki/Block_size_limit_controversy.

[4] This is why projects such as Fibre (http://bitcoinfibre.org/public-network.html) aim to provide high-speed links between certain locations.

To observe transaction inclusion and commit times on Bitcoin, we ran an experiment twice to allow for varying network conditions and growth of the network. Each experiment lasted ca. 25 h; the first was conducted in November 2016, the second in April 2017. We collected roughly 300,000 transactions in each experiment. It should be noted that websites like https://blockchain.info/unconfirmed-transactions reported high network load while the second experiment was being carried out, with 25,000–30,000 transactions waiting for inclusion.

We summarize the commit times (using 6-confirmation) we determined in Table 11.1. Note that they are significantly higher and more varied in the second experiment. Figure 11.2 plots the commit times for the two forms of transactions that are our primary interest. The blue curves refer to transactions that were a 'straight-accept', i.e. the parent transactions were known and the incoming transaction passed all mempool tests. The violet curves are the transactions that were orphans upon arrival.

Table 11.1 Summary of commit time distributions (in seconds) for orphans and straight-accepts during our experiments

Type	Min	Q1	Median	Mean	Q3	Max
Experiment 1						
Orphans	944	3096	4635	7582	8334	117,585
Accepts	676	2887	4234	5475	5901	150,123
Experiment 2						
Orphans	1293	4280	6337	34,912	51,352	174,516
Accepts	1165	3873	5364	18,417	19,286	171,566

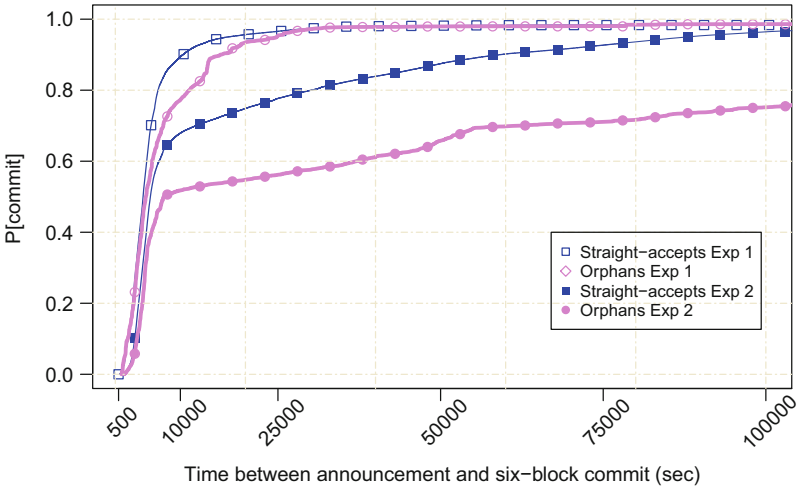

Fig. 11.2 Time between reception of transaction and commit. Note the logarithmic x-axis. © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

In an underutilized network, the theoretical median 6-block commit time should be around 3900 s: six blocks of 10 min or 600 s, plus on average half an inter-block time of waiting time for mining on a new block to start. In experiment 1, the median commit time for straight-accepts was 4234 s and the 90th percentile 9501 s. For experiment 2, these times were 5364 s as a median and 55,976 s for the 90th percentile. In summary, *even if waiting twice as long as the median time, more than 10% of transactions were not committed yet.* This is an important factor to consider when building an application based on the Bitcoin blockchain: median commit times are high, but individual commit times can be much higher.

We then did a number of analyses to examine delays and orphans further. In both experiments, orphans seem to be committed later than transactions that were directly accepted. However, the additional delay is much higher in the second experiment (where the network was under high load). In our first experiment, about 60% of orphans were included within the same time span as normal transactions. In fact, 31% of orphans took longer than 2 h to be included, 21% longer than 3 h, and 8% took longer than 5 h. For directly accepted transactions, these values were slightly different: 17% of them took longer than 2 h, 9.5% longer than 3, and 5% longer than 5 h. In our second experiment, roughly 40% of orphans had similar commit times as directly accepted transactions. The majority experienced very significant delays: the median was almost 20% higher, and the third quantile is more than 2.5 times as high as that for straight-accepts. We also note that only 1.2% of orphans and 1.6% of directly accepted transactions had not been included by the end of our observation period in experiment 1. In experiment 2, more than 20% of orphans had not been included (but almost all straight-accept transactions).

Factors other than the out-of-order arrival might still exercise considerable influence on commit times. We hence decided to investigate two further factors: transaction fees and locktimes. We first determined the number of transactions with a zero fee. This was always very low: for the straight-accepts, it was 74 and 12 transactions in experiments 1 and 2, respectively. The orphans *never* had a zero transaction fee. Figure 11.3 shows a box plot of transactions fees with the zero values filtered out. We can see that transaction fees are considerably higher in the

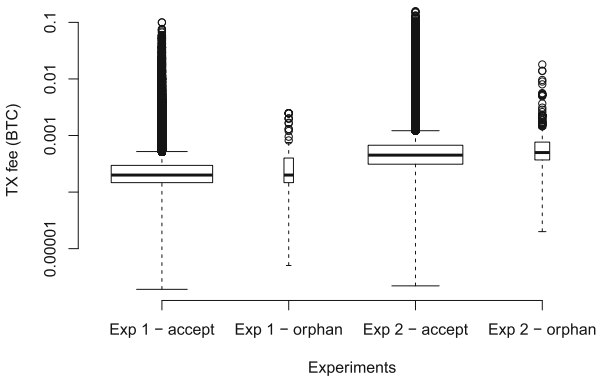

Fig. 11.3 Box plot of transaction fees by transaction category. Note the logarithmic y-axis. © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

second experiment, but there is no difference between straight-accepts and orphans in experiment 1. In experiment 2, orphans even have slightly higher fees. It is very unlikely that lower transaction fees are a cause for delayed commit of orphans.

We extracted the locktimes for our collected transactions and the locktimes of their parents. As our logger had not captured the full content of transactions arriving in the mempool (but only hash value and timestamp), we conducted this analysis only for those transactions that had been incorporated into the blockchain. The vast majority of transactions had no locktime set: in experiment 1, only 15% of straight-accepts and 12% of orphans had a value that was not zero. In experiment 2, the numbers were 23% and 17%, respectively. While this may signal an increase in the use of the feature, orphans *never* had locktimes beyond the observation window. Orphans in experiment 1 had locktimes that ended at least 3 h before the end of the observation window; in experiment 2 it was 6 h. In contrast, straight-accepts did have locktimes that extended considerably beyond the end of the observation window. In experiments 1 and 2, nearly 100% of transactions also had locktimes similarly near the end of the observation window. However, we found some decidedly optimistic locktimes on the order of 1.5–1.7 billion (block sequence number). With 10 min being the average time between two Bitcoin blocks, these transactions cannot be included before the year 30,166. The obvious limitation of our work here is that we do not know the locktimes of those orphans that were not included in the blockchain by the end of our observation period. Given the above results, however, we still feel confident to say that locktimes are not likely to be a decisive factor in commit delay of orphans.

Naturally, there may still be confounding factors in our study that we could not control for in this experiment. For example, we do not have information about node connectivity outside of our observation post, Australia, and could not determine the (ever changing) Internet routing constellation that the Bitcoin network is exposed to. Note that propagation times in the Bitcoin network have been investigated before. Our study suggests that it is worthwhile to revisit this topic.

11.6.2 Variation in Ethereum Transaction Commit Time

In this section, we first explain why Ethereum transactions are not guaranteed to be committed regardless of their validity. We then analyse if gas price, gas limit, and the network as factors affect commit time.

Recall the life cycle of individual transactions in the Ethereum blockchain from Section 2.2, depicted in Fig. 11.4. It starts with the submission of a transaction into the (virtual distributed) transaction pool across all miners. A transaction lifespan can be split into consecutive phases: (i) the announcement of the transaction in the system; (ii) the inclusion of the transaction in a newly mined block on some branch of the chain; (iii) the inclusion of the transaction in a block part of the main chain; and (iv) the commit of the transaction after sufficiently many confirmation blocks are subsequently mined.

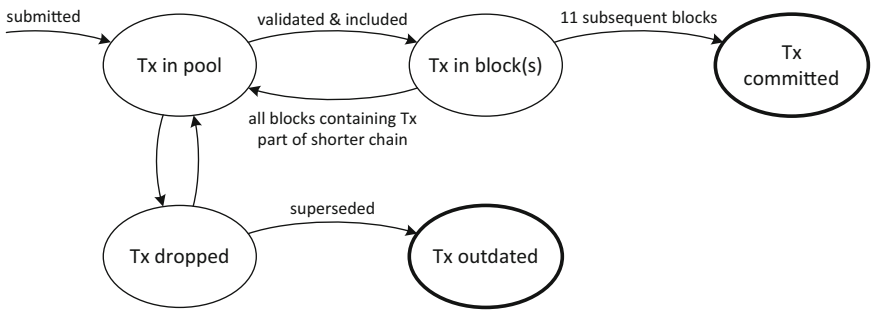

Fig. 11.4 Life cycle of an individual Ethereum transaction (notation: state machine; repetition of Fig. 2.7). © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

There is no certainty whether a particular transaction will eventually be committed or whether it will be *outdated*, in that it will be considered an invalid transaction forever. Moreover, it is impossible to know whether a transaction that is invalid in some state of the system will never be valid in a later state. More specifically, the aforementioned step (ii) is not sufficient to guarantee that a transaction *Tx* is permanently added to the blockchain: if the blockchain forks, then the block comprising the transaction may simply be discarded, in which case the transaction could be re-included later.

To put it differently: there are only two final states in this life cycle, namely, *committed* or *outdated*, and only these and inclusion in a block are observable transaction states for each client. In order to build a robust application on this basis, one needs to ensure that each transaction ends up in one of the final two states in a reasonable time. Otherwise the status of the transaction is, from the viewpoint of the client, undefined and unknown.

When a transaction is included in a block, it has been validated beforehand, i.e. its digital signature has been checked, as well as the validity of parameters like the nonce (sequence number of transactions relative to a given source account), and that there are sufficient funds in the source account. If all blocks that included the transaction become *uncles*—i.e. part of a shorter chain than the main chain—then the transaction goes back into the transaction pool. This may happen more than once, and, theoretically, there is no upper limit. While the transaction is in the pool, it may also be dropped. This is a local decision of miners, and it is impossible for any node in the network to know with certainty that all miners have dropped the transaction. Only when the nonce of the transaction becomes outdated, i.e. another transaction from the same source account with the same nonce got committed, can a node be certain that the transaction is invalid and will not be included in any valid block. Otherwise the transaction may resurface at a later point and get included in the chain.

Ethereum's transaction handling and inter-block time differ significantly from Bitcoin, and the chance of a chain fork occurring is higher. If a fork occurs, there is usually no certainty as to which branch will be permanently kept in the blockchain

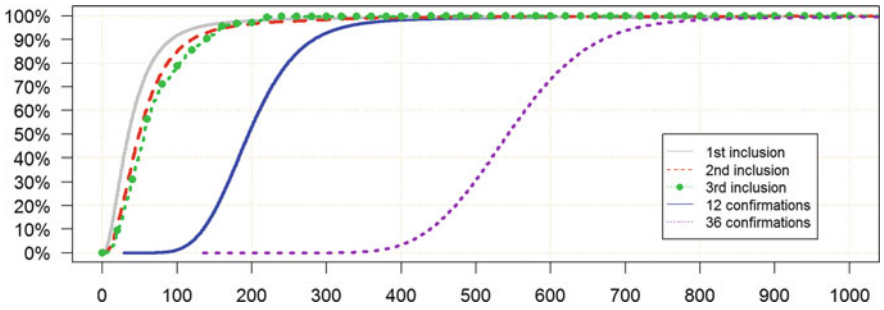

Fig. 11.5 Time (s) for first inclusion and commit (12 or 36 confirmations), as well as second and third inclusions of transactions that were previously included in uncles. © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

and which branch(es) will be discarded. In particular, transactions that were only included in uncles need to go back to the transaction pool. Before investigating the factors that cause commit delays, we investigate how fast transactions proceed from first inclusion to commit.

To empirically investigate transaction inclusion time in Ethereum, we collected data on *approx.* 6 million transactions over a 3.5-month period, discarding any short periods affected by network or system outages. The observations were conducted between December 2016 and April 2017. Figure 11.5 depicts the observed distributions of the time it takes for an Ethereum transaction to be included in a block and committed (using 12-confirmation, i.e. 11 subsequent confirmation blocks after inclusion, and 36-confirmation).

As shown in the figure, the inclusion times tend to follow similar curves. However, compare the slopes of the curves for first to third inclusion to the slopes for 12-confirmation and 36-confirmation: the latter are less steep, indicating the growing fraction of transactions that have to wait longer for a 'commit'. For a '12-block commit', the median time is around 200 s, and even the third quantile is not much higher. But the more blocks we require for a commit (say, 24 or 36 blocks), the more likely it becomes that a transaction needs (even considerably) longer than the median would suggest.

In contrast to Bitcoin, for the observed period the 90th percentile of commit happened significantly earlier than twice the median: at about 270 s for 12 blocks and around 650 s for 36 blocks. Still, while the curves converge towards 100%, they do not reach it within 1000 s. As a consequence, *applications sending larger volumes of transactions need to be prepared that some of these will not be committed in due time.*

Concerning transactions that become 'unincluded', however, we find that these are rare indeed. We observed that 113,122 first transaction inclusions (0.021%) were not permanent; and the same is true for 2602 second inclusions (0.0005%) and 41 of the third inclusions (0.000007%).

Ethereum has two user-defined parameters around the concept of gas, namely, the gas price and the maximum gas offered for including a given transaction. We

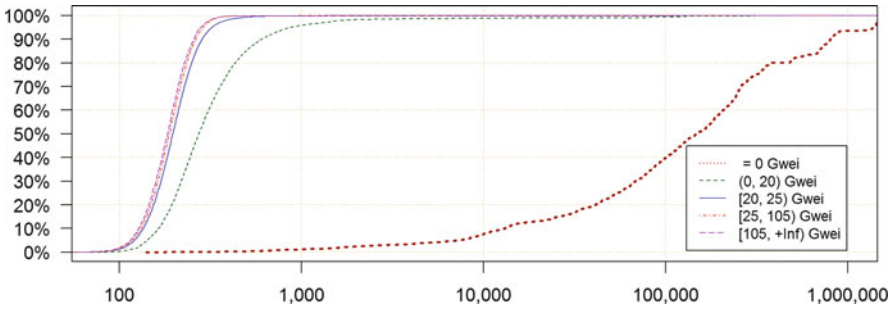

Fig. 11.6 Commit delay (s) for transactions based on gas price. Note the logarithmic x-axis. © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

proceeded to investigate how these affect the commit times. In particular, we were interested to see if it is possible to speed up the commit time by offering particularly high rewards for miners by setting a high gas price.

Based on our collected data, we analysed the effect of the user-defined gas price on the time it took for the transaction to be committed. Figure 11.6 depicts this relation for five bands of gas price (all in Gwei[5]): $[0, 0]$, $(0, 20)$, $[20, 25)$, $[25, 105)$, $[105, +\infty)$.

As shown in the graph, the higher the gas price in a given band, the less likely we observed long delays. However, we did not observe any meaningful differences from 25 Gwei onwards. At the time of writing in 2018, this observation is unlikely to hold true to the same degree: from anecdotal evidence, it appears miners behave more rationally. Finally, there is a sharp contrast between the 0-band and all other bands: the 0-band has significantly longer commit times.

A second user-defined variable around transaction fees is *maximum gas*, i.e. how much gas the execution of the transaction may use. We analysed its impact on commit delay. While we discovered individual transactions that were delayed due to an exceedingly high gas limit, our analysis was inconclusive: we could not find a strong correlation in any direction between maximum gas and commit delay. This remains an open question for now and warrants longer observation.

We were also curious to see whether the Ethereum network suffered from transaction reordering as we had observed it for Bitcoin. Ethereum does not link transactions in the way Bitcoin does, but every transaction has a running sequence number ('nonce') for each sender account. This sequence number starts from 0 and increments by 1 for each transaction sent from the same account. It is intended to provide an assurance that transactions from the same account will be executed in a particular deterministic order. However, it also means that a transaction with a nonce $n + 1$ cannot be included into the blockchain unless there is an already included transaction with nonce n—it is 'orphaned'. The transaction with the higher nonce will wait in the transaction pool until the arrival of a transaction with n as nonce.

[5] 1 Ether are 10^{18} wei.

We hence carried out an experiment that is similar in nature to our previous Bitcoin experiment. We analysed the commit times for *in-order* and *out-of-order* arrival of transactions during the same interval as for our second Bitcoin experiment, in April 2017. The total number of transaction announcements, which were also committed during this period, was 87,384. The number of transactions with out-of-order nonces was 5403 (6.18%). The commit time for both categories is shown in Fig. 11.7. The graph suggests that the commit delay for *out-of-order* transactions is almost doubled, compared to *in-order* transactions. To exclude the gas price as a confounding factor, we plot the gas price distribution for both categories, shown in Fig. 11.8. We did not find a significant difference in gas prices between two categories.

As with Bitcoin, it is hard to rule out other confounding factors that we cannot control for, e.g. Internet routing or overall network connectivity. However, our data allowed us some partial insight into the latter. We inspected transactions with nonce n that were announced *after* transactions with nonce $n + 1$ and compared these with *in-order* transaction announcements. Figure 11.9 plots the distribution of unique Ethereum nodes that we saw broadcasting the transaction before inclusion in the block. We found that delayed transactions were known to much fewer nodes. While

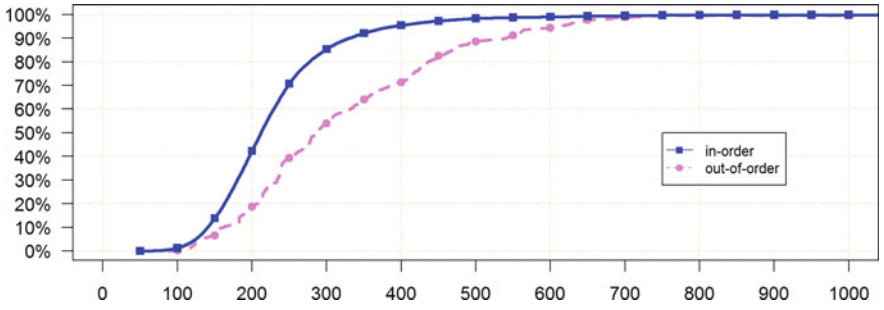

Fig. 11.7 Commit delay (s) for transactions based on ordering. © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

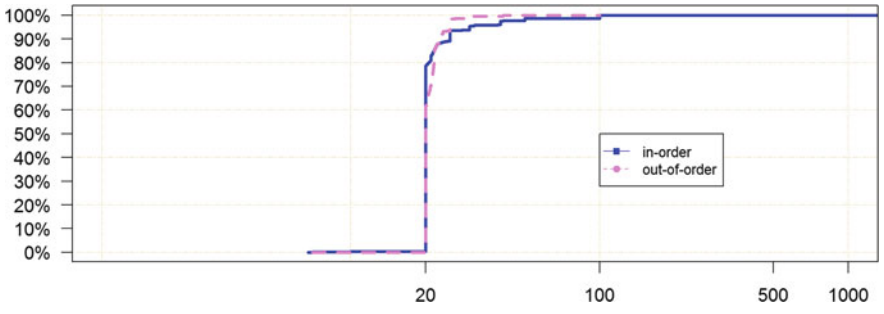

Fig. 11.8 Gas price distribution (GWei) for transactions based on ordering. Note the logarithmic x-axis. © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

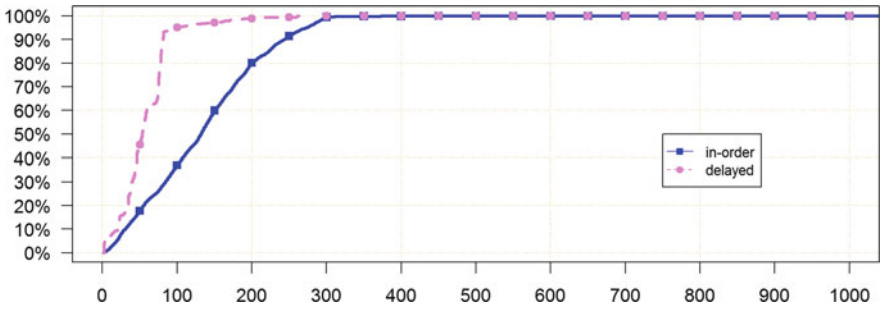

Fig. 11.9 Number of different peers from which in-order and out-of-order transactions arrive. © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

not conclusive, this provides first indications that network connectivity may have negatively impacted transaction propagation.

Ethereum has a second form of limit, the so-called gas limit per block. Unlike the gas price in a transaction, it is defined by the network of miners and applies to the *sum* of gas consumed by all transactions in a block. If the limit is lower than the gas required for a given transaction, the transaction cannot possibly be included. The development of the gas limit over time is readily available, e.g. on Etherscan.[6]

The rationale for the limit is to prevent denial-of-service (DoS) attacks on the network by limiting the amount of computation that can be done per block. Due to several DoS attacks against the network, a majority of miners on Ethereum agreed to lower the limit to *approx.* 500,000 gas temporarily—from October 15 to 17, 2016, according to Etherscan. The network still kept a low limit prior to and after these 3 days: from September 23, 2016, to November 22, 2016; with 1-day exception, the limit was around 2M gas. Around December 5, it returned to 4M gas. This limitation can negatively impact the inclusion of transactions which require a high amount of gas. This is not a hypothetical case: in earlier work, we deployed contracts using around 1.5M gas ourselves. However, simple transfers of assets should not be negatively impacted.

We hence chose to investigate whether we could find evidence for this hypothesis in our data. We analysed all transactions that happened before the DoS attacks and used block 2,303,121 as the pre-DoS cut-off block. We considered the amount of gas used for three different types of transactions: financial transfers, regular function calls to contracts, and contract creation.

Figure 11.10 shows the distribution of gas used for these transaction types. It highlights the gas limits mentioned above as vertical lines. No *financial transfer* transaction used more than 100,000 gas. This was an expected finding, as a financial transfer will incur 21,000 gas as base cost for any transaction, plus possibly a small amount for attached data: between 4 and 68 gas per byte (used, e.g. for a description

[6] https://etherscan.io/chart/gaslimit.

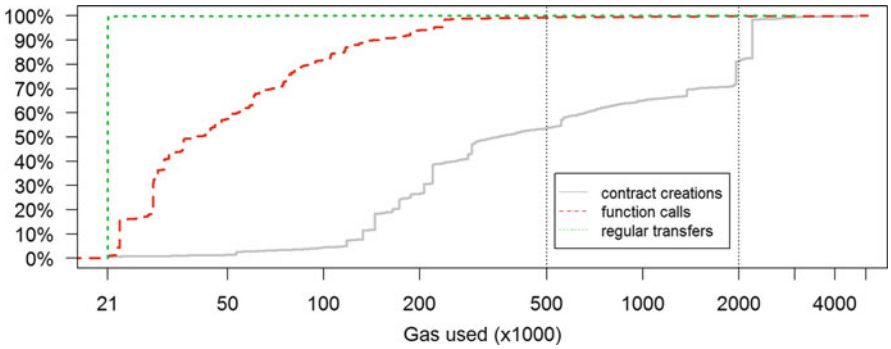

Fig. 11.10 Distribution of gas usage for different types of transactions, prior to DoS attacks. Dotted vertical lines show limits in response to the attacks. © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

of the transfer. As for *function call* transactions, 94% of them used at most 200,000 gas. Only 0.62% of the remaining function call transactions would not have been possible with the 500,000 gas limit. This contradicted a part of our hypothesis and highlighted that most of the functions that were in use were not highly demanding in terms of computation or storage.

However, when inspecting contract creation, we found that only 53.79% of all the contracts created before the DoS attack could have been created with the 500,000 gas limit, while 46.21% required more gas. This confirmed our hypothesis that many contracts would not have been deployable while the low block gas limit was in place. Even for the 2-month period where the network kept the block gas limit at about 2 million, 18.78% of contract creation transactions would have been impossible.

11.7 Aborting and Retrying Blockchain Transactions

One issue for a system designer who is building a blockchain-based system is that there is *no option to abort* a transaction. In this section, we propose a mechanism to artificially abort Ethereum transactions by superseding them with an idempotent or counteracting transaction. This abort mechanism can be useful if, for instance, the system observes that the transaction has not been committed within a specified time frame (as can be the case with, e.g. orphans). As such, the retry and abort mechanisms could be implemented to increase the user-friendliness or robustness of software clients or wallets.

Another motivation for abort is the accidental duplication of transactions, which we discovered thousands of times in our observation of Ethereum: the same transaction was submitted twice, often within seconds, but with different nonces, and the funds in the sender's account were insufficient for both transactions to execute. Seemingly the senders thought they were retrying the same transaction,

when really they created two separate transactions with the same parameters, except for the nonce. Instead of transferring the desired amount once, it would be transferred twice (if the balance was or became sufficient). This has also happened to individuals we know personally.

11.7.1 Aborting and Retrying Transactions in Ethereum

There are some options to achieve an effect that is similar to an explicit abort. In Ethereum, for instance, the system or user can issue a competing transaction from the same source account, i.e. another transaction with the same nonce. Assume user Alice transfers 1 Ether to Bob by issuing transaction Tx_i with nonce i. After an acceptable time frame, e.g. 10 min, has elapsed and Tx_i has not been committed, Alice wants to abort Tx_i. She then submits a new transaction Tx_i', with the same nonce i as specified in Tx_i and a higher transaction fee in order to increase the chances for Tx_i' to be included. For this transaction Tx_i', she does not want to spend more Ether than necessary; thus, she sets the transaction value to 0 and her own account as receiver. Once Tx_i' is committed, Tx_i is superseded by it and becomes outdated. If, in the meantime, Tx_i were to succeed, Tx_i' becomes outdated. This is acceptable, since that was the original intent.

Alternatively to aborting, Alice can 'retry' Tx_i by submitting Tx_i'' as follows: the fields in Tx_i'' contain the same data as in Tx_i, including nonce i—except Alice offers a higher fee for it. Therefore, the hash and digital signature of Tx_i'' will be different from Tx_i, and thus it will be perceived by the miners as a separate transaction. If Alice tried resending Tx_i without any changes, hash and signature would be the same, and the miners would not consider it any differently—unless they have previously dropped Tx_i. In the latter case, the reasons for dropping Tx_i might not have changed, and thus the same would likely happen again. If either Tx_i or Tx_i'' succeeds, the respective other transaction would become outdated and invalid, since they both have the same nonce i.

11.7.2 Aborting and Retrying Transactions in Bitcoin

The Bitcoin blockchain does not offer transaction abort. In a German newspaper article from late 2017, the author described that he 'lost' BTC[7] worth several hundred Euros, since he did not offer a transaction fee, and his transaction had not been included for more than 2 weeks. His wallet application did not offer options to abort or retry the application, and simply reported his account balance to be zero. In cases like that, we believe the following method should work.

[7] BTC is the currency code for Bitcoin's cryptocurrency.

Say, Alice wants to transfer 5 BTC to Bob. She previously received 2 BTC from Charlie, as output 0 (abbreviated as #0) in Tx_1, 1 BTC from David as #0 in Tx_2, and 4 BTC from Erin as #1 in Tx_3. Her virtual account thus holds 7 BTC. To achieve the transfer, Alice creates transaction Tx_{orig} that has two inputs: Tx_1 #0 (2 BTC) and Tx_3 #1 (4 BTC), so that Tx_{orig} has a transaction volume of 6 BTC. Alice then adds two outputs: #0 with 5 BTC to Bob and #1 with 0.99 BTC to herself. Tx_{orig} thus offers a transaction fee of 0.01 BTC, and subsequently her virtual account will hold 1.99 BTC.

Now, say the commit of this transaction does not happen within Alice's timeframe of 6 h and Alice wants to abort. Since each input can only be spent once, Alice can achieve that by submitting Tx_{abort} with the same inputs as Tx_{orig}, but as single output #0 she specifies 5.98 BTC to herself (thus offering a transaction fee of 0.02 BTC). If either Tx_{orig} or Tx_{abort} succeeds, Alice's account is not in limbo, and she can continue to use the network as normal.

As an alternative to abort, Alice can re-attempt the transfer with Tx_{retry} as follows. The inputs are the same as in Tx_{orig}, output #0 stays at 5 BTC to Bob, but output #1 is changed to transfer 0.98 BTC to herself. Tx_{retry} thus offers a higher transaction fee of 0.02 BTC, and if Tx_{retry} succeeds then Tx_{orig} becomes outdated.

11.7.3 Experiments for Aborting Transactions in Ethereum

We tested the above method for abort on the public Ethereum blockchain for three scenarios: (i) a transaction does not get included in the usual period of time; (ii) a client changes its mind and decides to roll-back the issued transaction; and (iii) a transaction is in indefinite pending state due to insufficient funds. We describe these below in more detail.

Abort Experiment 1 In order to test the situation where a sent transaction does not get included in the usual timeframe, we submitted 100 transactions that *underbid* the market rate. Specifically, we assumed the average gas price from the previous day (December 1, 2016) as market rate (mr) and submitted ten transactions each for different prices, which are 0, $0.1 \times mr, 0.2 \times mr, \ldots, 0.9 \times mr$. As cut-off time, we rounded up the 99% percentile from our earlier experiment (cf. Fig. 11.7) to 10 min. If the transaction had not been included then, we submitted an abort transaction Tx_{abort} as described above, with the same nonce but at full market rate mr, target `0x0`, and value of 0.

The results are shown in Fig. 11.11. Surprisingly, most transactions were accepted by the network. Six out of ten transactions with either 0 or $0.2 \times mr$ were accepted. In addition, only two out of ten transactions with $0.1 \times mr$ were accepted. All of the 16 timed-out transactions were successfully aborted with our Tx_{abort} mechanism described above.

Abort Experiment 2 For this experiment, we assumed a client that underbids the market fee and changes its mind regarding an issued transaction. As in the previous

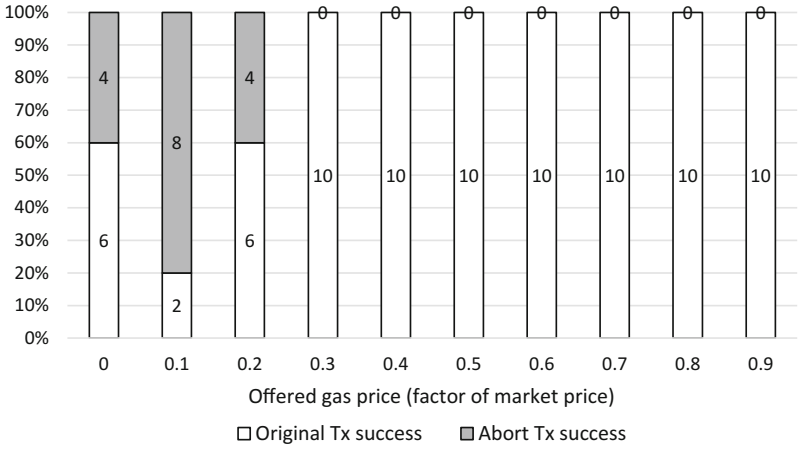

Fig. 11.11 Underbidding market fee and automatic abort after 10 min if the original *Tx* was not included. © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

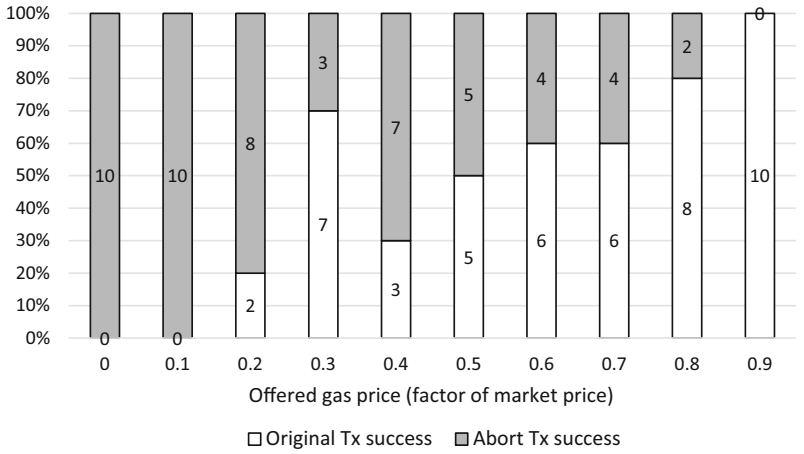

Fig. 11.12 Underbidding market fee and automatic abort after 3 min if the original *Tx* was not included. © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

experiment, we sent another 100 transactions with gas prices as above, i.e. 0, $0.1 \times mr$, $0.2 \times mr$, ..., $0.9 \times mr$ for ten transactions each. Rather than waiting for 10 min, we set the timeout value to the target median for Ethereum transaction commit, i.e. 3 min.

The results of this experiment are shown in Fig. 11.12. A much higher percentage of transactions were not included in a block after 3 min, in comparison to Fig. 11.11 with 10-min timeout. As before, 100% of Tx_{abort} succeeded. Interestingly, all of them were included in a block after 3 min. In 2 out of the 100 cases, the 3-min timeout for the original transaction was reached, Tx_{abort} was sent, but the original transaction Tx_{orig} still won the race and got included and committed in

the blockchain. Thereby, Tx_{abort} was outdated. As stated above, this is a possibility that clients should be prepared for. The reasons for such a situation can include (i) processing time in our client when preparing the Tx_{abort}; (ii) broadcast delays or other network effects where the winning miner does not receive Tx_{abort} before including Tx_{orig}; or (iii) non-rational scheduling of transactions in the pool, where no preference is given to the transaction with the higher fee.

Abort Experiment 3 In this last experiment, we submitted two transactions, creating a situation that corresponds to faulty inputs from a user (or user's program). We have observed such behaviour during our live observation of the public Ethereum blockchain. To replicate it, we submitted two transactions, Tx_1 and Tx_2, as follows. Assume that the last nonce for the sender address was n and its account balance k. Then we create Tx_1 with nonce $n + 1$ and value $\frac{1}{1000}k$ and Tx_2 with nonce $n + 2$ and value $\frac{999}{1000}k$. For both transactions, we set the gas price to $0.7 \times mr$. Due to the nonce, Tx_1 must be included before Tx_2. However, due to the positive gas price, the account balance resulting from the inclusion of Tx_1 is insufficient for Tx_2.

Finally, we submit Tx_2, wait 5 s, and then submit Tx_1. This gives Tx_2 the chance to get broadcast before Tx_1 is known to any node, including our own. This procedure is needed so that the client submits Tx_2 to the network; since geth is not aware of Tx_1 and its contents when we submit Tx_2, it broadcasts Tx_2. Otherwise, it might detect the insufficient balance and not accept Tx_2.

Once Tx_1 has been included in a block, Tx_2 is invalid due to insufficient funds. However, this does not always get checked, and hence Tx_2 may remain in the transaction pool for a long time. In fact, if another transaction deposited funds into the sender account, Tx_2 would become valid and be executed. This, again, is behaviour that we observed. Here, we send a Tx_{abort} with the nonce $n + 2$, to abort Tx_2.

We ran this experiment until we had submitted Tx_{abort} 100 times. All 100 submitted Tx_{abort} were successful. We measured the time it took for Tx_{abort} to be included in a block (first inclusion) and plotted that as shown in Fig. 11.13. The median for those times was 45 s and the maximum 230 s.

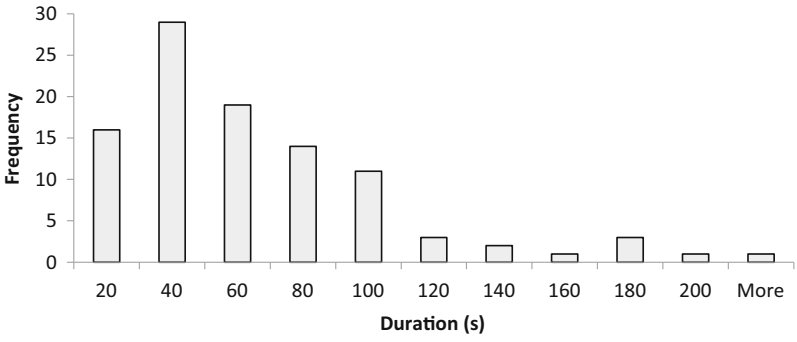

Fig. 11.13 Abort duration histogram, from experiment 3. © 2017 IEEE. Reprinted, with permission, from Weber et al. (2017)

Our experiments support the hypothesis that transactions can be aborted with our proposed method. Although it would be better to have explicit abort mechanisms for blockchains, this is a fall-back method for certain applications to address commit delays that are due to some of the factors we have described in Section 11.6.2.

11.8 Summary

We started this chapter with a broad discussion on the impact that using blockchain as a component can have on dependability and security properties. In short, confidentiality can be harder to achieve, due to the replication of the data structure to the whole network; integrity is blockchain's strong suit; in terms of safety, the picture is less clear; maintainability requires planning and governance; and availability and reliability features are high for reading/receiving, but potentially low for writing/sending.

To give a clearer picture of the write/send availability and reliability characteristics, we studied the public Bitcoin and Ethereum networks. For Bitcoin, we found that even if waiting for a transaction commit twice as long as the median time, more than 10% of transactions were not committed yet. For Ethereum, this was less common, but still above 1%. This is important when building an application based on public blockchains: commit times vary significantly and can take significantly longer than in common cases.

Finally, we discussed methods for transaction abort and retry, which are not built-in functions of blockchain clients. Applications can use these methods to handle transactions that take unusually long.

11.9 Further Reading

This chapter is partly based on Weber et al. (2017) and draws on earlier ideas from Anderson et al. (2016).

As stated in the beginning of the chapter, we did not cover security infrastructure or cryptography. A number of books discuss these points in detail, e.g. Bashir (2018).

In this chapter, we refer to a few seminal works, specifically the Clark–Wilson security policy model (Clark and Wilson 1987), and the taxonomy of dependable and secure computing by Avizienis et al. (2014). The alternative definitions of safety are described in Lamport (1977) and Alpern and Schneider (1985). Finally, the CAP theorem (Fox and Brewer 1999) indicates that there is inevitably some trade-off between consistency, availability, and partition-tolerance for distributed databases.

The Ethereum yellow paper (Wood 2015–2018) specifies gas costs for various operations and describes the function of block gas limits.

As mentioned in the previous chapter, live statistics about the public Ethereum chain, including inter-block times and the influence of the gas price on transaction inclusion times, are available at https://ethstats.net/ and the ETH Gas Station (https://ethgasstation.info/). ETH Gas Station also gives recommendations for gas price settings, relative to desired inclusion times; these can also be accessed through an API. From these recommendations it appears that, at the time of writing, miners now react more to gas prices and the network is less likely to accept transactions offering no fee than it did when we conducted our experiments.

An earlier investigation on propagation times in the Bitcoin network has been conducted by Decker and Wattenhofer (2013).