Annals of Emerging Technologies in Computing (AETiC)
Vol. 3, No. 4, 2019

Research Article

Consideration of Quality Attribute Tradeoffs of the Blockchain Pattern in the Software Development Process

John M. Medellin and Mitchell A. Thornton*

Darwin Deason Institute for Cybersecurity, Southern Methodist University, Texas, USA
JohnMedellin@Verizon.net; Mitch@SMU.edu
*Correspondence: Mitch@SMU.edu

Received: 1st September 2019; Accepted: 17th September 2019; Published: 1st October 2019

Abstract: The Blockchain (BC) design pattern has many variations and is a concept that is anticipated to lead many implementations in the years to come. The number of choices for a BC implementation continues to increase since new design and implementation patterns and applications are emerging. This increasing number of design patterns enables correspondingly increasing tradeoff opportunities at every evolutionary round of architecture elaboration. Key components of a BC include network nodes, blocks, and consensus methodologies. These components all possess critical characteristics that can be designed and implemented in a variety of different ways. A central thesis here is that the choice of the design methodologies has direct and varying impact with regard to resulting quality attributes such as performance, security, and availability. We describe the use of a tradeoff matrix during the initial design phase of a development cycle that identifies the quality attributes to be evaluated when designing software systems comprising a BC. We hypothesize that consideration of the quality attributes at this initial design stage via the use of the proposed tradeoff matrix enables designers to meet requirements more efficiently and accurately. This hypothesis is tested and the use of the tradeoff matrix is demonstrated by creating a consensus algorithm whose performance is evaluated through a simulation that compares the behaviour in a "bare-metal" versus a Cloud-based environment. This simulation approach drives the usage of one of the quality tradeoff parameters in achieving a more optimal solution.

Keywords: ***Blockchain (BC); Consensus; Quality Attributes; Design Tradeoffs; Software Development***

---

1. Introduction

The concept of replacing a centralized and authoritative record of approved exchanges of services or data with a decentralized record has many desirable features. Some of these include fault tolerance, enhanced security, and an ability to verify that such approved exchanges, or "transactions," are indeed valid. While the concept of using a redundant and decentralized transaction ledger as compared to a single authoritative ledger is somewhat simple, the details of implementing such a scheme can become technically challenging. A modern and popular

John M. Medellin and Mitchell A. Thornton, "Consideration of Quality Attribute Tradeoffs in a Blockchain Software Development Process", *Annals of Emerging Technologies in Computing (AETiC)*, Print ISSN: 2516-0281, Online ISSN: 2516-029X, pp. 15-27, Vol. 3, No. 4, 1st October 2019, Published by International Association of Educators and Researchers (IAER), DOI: 10.33166/AETiC.2019.04.002, Available: http://aetic.theiaer.org/archive/v3/v3n4/p2.html.

implementation was recently disclosed in the form of a collection of methods and data structures known as "Blockchain" (BC). The BC concept has enabled technology designers, providers, and users with a means to utilize a decentralized ledger that in turn offers attractive benefits such as protection from false or malicious transactions and enhanced fault tolerance. While BC was originally proposed to support a publically accessible ecosystem based upon a digital crypto-currency, the concept of a distributed record of transactions versus a centralized record has rapidly gained popularity for a variety of other applications.

As the BC concept has rapidly evolved, the community of technology providers has begun to classify specific use-cases based upon the application that is supported by a BC-based solution. For example, the initially disclosed application that supports a crypto-currency is an example of a "public" BC since it is desirable for any arbitrary entity to become a BC community member and to engage in crypto-currency commerce. That is, public BC members are free to join or leave the ecosystem of their own volition. The use of a BC to guard against counterfeiting in a supply chain is an example of a "permissioned" BC since valid participants in the supply chain are given permission to join the BC community and they are not necessarily free to join or leave the community. Such permission may be granted by a centralized authority, or by some other means such as the evaluation of a committee or subset of the existing community. The third, and most restrictive form of BC, is the so-called "private" BC. A private BC is one wherein the community that can access, utilize, and modify the BC is defined by design and closed. Furthermore, private BC community members generally do not have the ability to leave the community and they must support the usage and maintenance of the BC. At a lower level of abstraction, the BC can be considered to consist of a replicated data structure in the form of a hash-chain of abstract data types and a collection of methods for augmenting, modifying, and accessing records within the structure.

From the point of view of considering public versus permissioned versus private BC implementations, the details of the underlying algorithms and data structures can vary quite considerably although the overall purpose of the BC remains the same; namely to provide an infrastructure wherein the community can maintain a distributed ledger of transactions. This variation in implementations affords a designer with a large number of different design options to choose among. An experienced designer must still consider the far-reaching implications of selecting one course of action over another.

From its roots, deep in distributed operating systems and databases through current implementation, the pattern for information hiding through encryption and sharing of that information with other parties has continued and is continuing to evolve through requirements that are sometimes unknown until implementation. The pattern has been made famous by the implementation of Bitcoin, a cryptocurrency. In a primary objective, the pattern allows for transferring value between the peers through consensus and encryption of results in a distributed event ledger.

However, in addition to exchanging value, the pattern is also very useful for enabling methods of sharing secrets or other information between participants. The application of BC to support secure information sharing has been adopted by researchers and industry advocates for transmitting sensitive or private information between trusted parties in a group [1]. An example of this particular sharing is the transmission of PKI (private key infrastructure) keys in a network where only the intended receiver can decode such a transmission and by virtue of the BC pattern cannot repudiate (negate receipt) of that specific information. Furthermore, a block is mathematically bound with subsequent blocks through progressive hashing. The cumulative effect of this is to make the transaction a permanent one in the event registry resulting in an immutable transaction ledger.

This objective of this paper is to segment the key software quality attributes and associated tradeoffs a designer should consider when architecting a solution with this pattern in mind. We discuss our assessment of these key tradeoff patterns and provide a matrix that identifies the attributes that should especially be considered during software design as originally outlined in [2]. It should be further noted that these key attributes are only considerations and should preclude the consideration of all software quality attributes for any solution design. Here we provide detail and

depth with regard to the discussion in [2]. Additionally, we include an example based on BC consensus algorithms. The paper concludes with an example of how to apply the matrix in selecting a consensus algorithm; a very critical component of any BC architecture.

2. Related Work

Many variations have emerged since the disclosure of the famous Nakamoto [3] paper that served as the precursor to the famous Bitcoin series of cryptocurrencies. This first paper on the nature of exchange of value as enabled through use of the BC design pattern has been widely adopted and now constitutes several billion currency units of value in over 700 crypto exchanges around the world with Bitcoin being the most famous.

Since the work of Nakamoto in 2008, most of the implementations of BC support cryptocurrency applications. However, in recent studies, the design pattern has begun to be adopted as a means to enable data transmission while preserving privacy. Indeed, many technology implementations have begun to look at this design pattern as a means that supports and protects private exchange of information between interested parties [4, 5].

A significant difference in implementation patterns exists when a private BC is implemented versus a public BC [6]. In a public BC environment, parties do not necessarily have to know or trust each other in order to exchange value. Rather they can engage in a transaction by creating a candidate transaction that is cryptographically validated by other members in the network known as miners. Miners are a subset of BC community members who are not one of the transacting parties and who are incentivized to validate candidate transactions by receiving units of cryptocurrency if they win a contest based upon the first member to validates the candidate transaction first. The validation is performed through the solution of a computationally hard task or "puzzle" that is ingenuously devised to also validate the transaction. Since the miners must perform computational "work" to solve the puzzle and the solution of the puzzle inherently verifies a candidate transaction, this approach is often referred to as a "proof of work" approach. Once a miner offers the community a potential puzzle solution, the community is obliged to validate that the solution is correct. Thus, the puzzle must be computationally hard to solve but computationally easy to validate. A common example of such a proof-of-work puzzle is the problem of inverting a cryptographic hash function. It is known to be computationally hard to find a hash key given the hash value and function; however, it is very easy to simply compute a hash given a candidate key value. To deem a candidate puzzle solution as being valid, some predetermined percentage of BC community members must all agree that they correctly validated the proposed puzzle solution; or in other words, consensus must be achieved. Only after consensus is achieved will the miner who proposed the puzzle solution receive the cryptocurrency reward and the candidate transaction be committed to the distributed ledger. The problem of computing consensus in a distributed asynchronous environment is an instance of a well known and studied problem in distributed processing systems and the particular consensus method used can greatly affect the performance of a BC implementation.

In contrast, in the context of a private exchange of data, the parties wish to know and trust each other before the exchange happens. Some examples of private BCs include distribution of sign-on credentials [1] or authentication of agents delivering content on a home [7]. In these cases, the "value" to be exchanged is a secret or information that is only known to the sender and receiver [8] and potentially to other trusted parties in the closed network of operation for the BC. In these cases, the BC must utilize methods that guard against various cyber threats such as masquerading man-in-the-middle (MITM) attacks or MITM eavesdropping. These BC design objectives affect the types of methods that underlie a BC implementation and are currently regarded as open areas of research.

As previously mentioned, significant amounts of intellectual capital have been expended on the design and implementation of public BCs. However, our focus in our design example is a private BC to be used as a means to guarantee key properties of secure transmission models. We have written about usage of this design patterns in previous work and have focused mostly on the consensus architecture requirements and implementation [9].

Our previous work was dedicated to modeling different consensus algorithms and their impact on resource consumption. Most of the literature compares alternative consensus algorithms to the Byzantine General's Problem [10] since it has become a pseudo-standard for measuring performance characteristics in newer consensus algorithms. In this work however, we expand the scope of our discussion to include the other aspects of the BC architecture requirements in relation to private exchanges. In a later section we discuss the block, smart contract, network, consensus and cryptography aspects.

3. Design Considerations and Architecture Requirements

There is some variability in the definition of components required for a successful implementation. However, there is general agreement that the components required for the pattern to work include the definition of a BC community, the supporting BC data structures, and the supporting BC algorithms that are used to create, update, modify, and augment the BC data structure as well as those algorithms that are used for other community services such as the consensus method.

The network of participants or the BC community is a group of "nodes" that are typically individual processes running on within the same host or on different hosts that are all interconnected via a network. In this context, "network" refers to the group of members that are enabled to operate and utilize a BC [11]. Nodes of the network can utilize subsets of BC functions. For example, in a crypto currency application a node may simply be a consumer and do nothing more than engage in transactions. Alternatively, a node may be a consumer that also participates in maintaining a valid copy of the ledger, but that does not participate in mining. In the extreme case, a node may be a process that can perform transactions, maintains a ledger, and participates as a miner. In other BC implementations such as permissioned or private BCs, the BC community may have different classes of members. However, in all cases, a BC community member or node does have some capability in terms of performing transactions and can perform the basic operations of exchange and validation [12]. The value of a BC approach is in the maintenance of a distributed ledger that removes issues associated with a central authority that can become corrupted or malicious. The "distributed" aspect of the ledger lies in the fact that many different nodes maintain individual copies of the ledger that are forced to all be consistent via the methods and polices comprising the BC implementation. Thus, a BC community should be suitably large to ensure it is effective in terms of providing security, fault tolerance, or other similar properties.

A single instance of the distributed ledger is a data structure that is in the form of a linked chain of "blocks" wherein each block is comprised of a header and a list of abstract data types (ADT) that each represent a transaction. As transactions are validated, they are accumulated into a current block and when that block reaches a certain size, or a certain amount of time has elapsed, it is added to the chain or previously committed blocks and all BC members who maintain a transaction log update their individual ledgers with the newly committed block. Instead of address pointers, the blocks are linked through the use of hash pointers resulting in a hash-linked list of blocks where each block is comprised of a header and a list of transactions in the form of individual ADTs. Because the BC pattern is being applied to many different applications and among different types of communities, the specific content and specifications of an actual block is a design attribute itself and can vary [13]. A block is chained to a prior block by reference to the prior block via a hash pointer. While the specific ADTs within a block can vary, they typically are comprised of a header that contains the block pointer and a Merkle tree root as shown in Figure 1. A Merkle tree is a binary tree that is comprised of hash pointers rather than address pointers. Each transaction is a leaf in the Merkle tree and the content of the transactions is hashed together to form an inherent Merkle tree. The purpose of Merkle tree is to support immutability as well as providing an efficient means to search for a transaction within a block.

While we have thus far considered the atomic event that is recorded in a ledger to be a transaction, it is possible to store any desired type of data in the distributed ledger. The requirement is that the proposed data to be stored is in a form that can be validated by all BC community members while also not necessarily being explicitly reveled. In addition to storing and recording data, such as

the record of a transaction, it is also possible to store a method in a BC. Because items stored in a properly implemented and functioning BC are immutable, when these items are in the form of an agreed set of actions between two parties, they essentially obey the legal definition of a "contract." That is, if two nodes agree upon a set of actions and that set of actions is validated by the BC community to be those that were originally agreed upon, and furthermore, a record of these actions are stored in the immutable BC data structure that also guards against nonrepudiation, then a contract has been executed among the two parties that originally proposed and agreed upon the set of actions. This type of object, when used in a BC, is referred to as a "smart contract." A smart contract is a reference to a set of procedures that the block will provide, and in some cases will cause to be executed, when the conditions of the contract are met [14]. The scope and specifications of a smart contract are a design consideration.

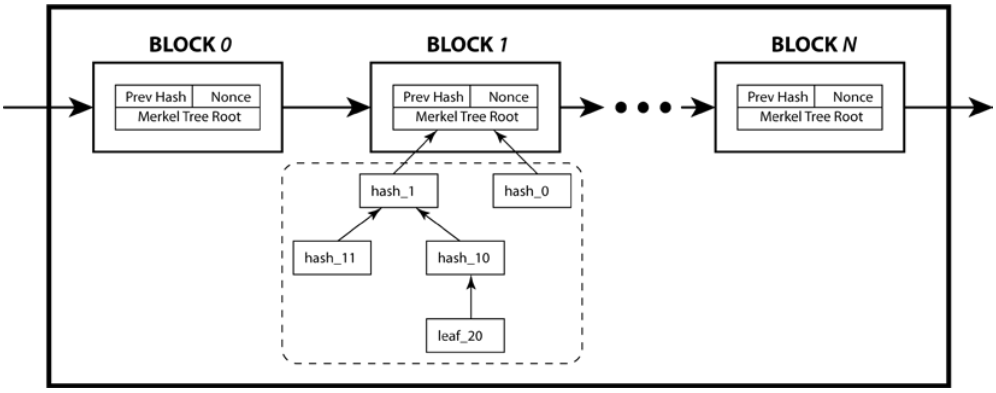

Figure 1. Diagram of Blockchain Architecture with Partial Detail

Smart contracts can be implemented as a block that defines a set of procedures in the actual data payload or a conditional reference to another part of the BC that contains other types of data. The smart contract, when invoked by a special block type, will produce a result that operates on the input data of that another block. A smart contract may contain the algorithm for processing the additional data or may reference a specific location for those instructions [15].

A BC requires several different types of algorithms to function properly. Some of these include hash functions both for the construction of the structures as well as possibly serving as sources for a mining puzzle. Algorithms are needed to maintain the ADTs, to maintain valid replications of the ledger, to update invalid copies of the ledger, to support the consensus model, to ensure the privacy of sensitive data, and a host of other support functions. In most BC implementations, it is desirable to encrypt portions of or the entire ledger, thus cryptographic methods and support functions such as nonce generators are important design considerations [16].

While there are other considerations, the final one that we wish to describe here is the consensus model. Broadly, consensus is the process whereby the participants agree in adding the new block to the chain [17]. An important aspect of the BC pattern is the avoidance of a centralized authority that may become corrupted either through a fault or via some malicious intent. For this reason, it is important that the model be computationally distributed rather than a naïve approach such as a centralized voting accumulator as such a centralized consensus model would negate the benefits sought through the deployment of a distributed ledger. The consensus model provides the method whereby sufficient proof of validity of a transaction is provided by a given set of node in order to append the block and commit it to permanent storage in the chain. Two of the most common consensus methods at this writing are based on Proof-of-Work (PoW) [18] and Byzantine Fault Tolerance Algorithms (BFTA) [19].

The consideration of the design of the consensus model must necessarily involve aspects of the application. For example, if the BC community consists of nodes that run on computationally lightweight CPUs, the consensus model must not require excessive CPU cycles to complete. Alternatively, if performance is the prime factor, then the consensus model should be one that rapidly

completes. Because consensus is known to be challenging, it represents one of the most important design considerations in almost all BC implementations.

3.1. Key Variability Parameters by Architecture and Security Ramifications

Each of the architecture patterns has a variety of choices that ultimately influence the achievement of objectives in a BC implementation. The variability of the major components and their respective impacts are summarized here with an emphasis on security.

The architecture of the block architecture is a major influencer on the objectives. On one hand, the block needs to be sized appropriately in order to include the necessary data. On the other however, if the block is too large the impact is felt in encryption computation and latency of transmission [18]. Depending upon the implementation, a very large block size can correspondingly increase the threat surface.

The nature of the smart contract has a very large impact on both performance and attack resilience. By including a smart contract in the implementation complexity and additional overhead is acquired [12]. If the contract is wholly contained in the design then by definition, some of the prior blocks will need to be modified. A wholly contained smart contract is a type of block that is mutable as it processes and receives the effect of transactions. If the contract has a reference to a third location, the attack surface is expanded, performance could be impacted by additional steps to be taken in retrieval and update of external entities. Finally, for subsequent block encryption, the result of a smart contract action must be taken into consideration so that the cryptography rules can continue unimpaired.

Network variability can be abstracted as the set of rules for adding, processing, and deleting nodes that interact with the BC. This might seem to be the most secure but it is where the attacks known as Impostor and Sybil can penetrate [8]. The variability in a private BC affects the usability of the method and can thus affect the integrity of the privacy of the BC community. A strict adherence to high trust generally necessitates that high complexity is present in most cases. The additional overhead that allows for high trust needs to be balanced with the usability of the B.

The consensus model represents an area of high variability in the implementation of a BC design. There are dozens of consensus models reported in the literature that have varying levels of both computational overhead and complexity required to ensure agreement without tampering. Several summaries of consensus models have been published including those in [20, 21].

Most BC patterns will use methods of cryptography and these are well documented [15]. The biggest issue identified has been the potential for quantum computing usage to break those algorithms. Several studies exist on post-quantum cryptography and we strongly suggest incorporating some of those initial rules into the design [22]. Post-quantum BC is currently an active area of research.

3.2. Software Architecture Quality Attributes

The Software Engineering Institute (SEI) has published guidelines for designing software systems with emphasis on architecture [23]. These guidelines are summarized and repeated here as follows:

1. Availability; the system is present and ready for interaction.
2. Modifiability; the degree to which the system can be reasonably altered.
3. Testability; the system has facilities that enable the validation of results.
4. Interoperability; the ability to interact with other systems.
5. Security; resilience to cyber-attack.
6. Performance; achieves targets, typically in terms of meeting timing deadlines.
7. Usability; enabling the use of the system for designated purpose.
8. (Optional) Extensibility; the ability to extend system functionality beyond its original scope.

Within the SEI framework, these represent key attributes in determining the quality of a BC software system. These attributes typically involve tradeoffs and cause a designer to carefully consider requirements to achieve a pseudo-optimal design. A classic example is performance versus security. Performance enhancements are typically achieved at the expense of increased processing conditions that in turn extend the threat surface area. In the next section, we address how these attributes and the corresponding and inherent tradeoffs interact with respect to usage of the BC design pattern.

4. Quality Attribute Tradeoffs

Software quality attributes drive design decisions that shape the final BC solution [24]. Consideration of general quality attribute tradeoff mechanisms while also comparing them to tradeoffs that are more prevalent in the BC design pattern is the goal of this section.

4.1. Architecture Design and Evolution Implies Quality Attribute Tradeoffs

Architecture design implies tradeoffs between quality attributes in order to deliver a solution that reasonably complies with stated requirements. Quality attributes are typically in conflict with each other, for example, the testability attribute which includes verbose output might be in conflict with the security attribute which aims at hiding the attack surface (verbose output implies additional software that may be turned on to describe the current state of processing but this additional software could also be attacked). Another example is the conflict between interoperability and performance. If a system is designed with very high level of interface abstraction and cohesion it might cause the system to use more cycles and impact performance requirements. These tradeoffs are even more specific for BC design and are discussed in the next section.

4.2. Key Blockchain Architecture Tradeoffs

Based upon a consideration of the BC architecture and components described previously, we list some of the dominant tradeoffs that should be considered in a BC architecture design. They are complemented further from the list provided by [25, 26, 27].

1. Storage versus Computation: the amount of storage required by the block will affect the computation requirements; a larger block will require more computation to encrypt since there are more elements to encrypt. The design tradeoff in this case is *Usability versus Performance*.
2. Anonymity versus Trust: if the parties do not know each other they most likely need to verify proper identities through the use of public/private keys. This additional step will require a different design pattern. The design tradeoff in this case is *Interoperability versus Security*.
3. Incentive variations: by definition, the pattern requires distributed validation before blocks can get appended. Participants are usually incented to validate candidate blocks in order to carry out this function. In a public BC, this usually means allocation of value to those participants that, in turn, implies tracking value in the network. Similarly, in a private BC, the incentive schemes vary, in the simplest form it could mean that the participant must validate prior blocks before theirs gets appended. The design tradeoff in this case is *Extensibility versus Usability*.
4. Degree of Distribution: In traditional Nakamoto-inspired BC patterns, the participants keep replicated copies of the BC. However, there are different BC patterns that may require that smart contracts reside offline, on the BC in a centralized or other node. The design tradeoff in this case is *Interoperability versus Extensibility*.
5. Scalability versus Latency: These two attributes are closely tied to performance. In most cases, systems are designed for fulfilling a certain return time to users that is specified in

requirements. When those limits are exceeded by larger scale (more volumes) the processing of blocks begins to lag, in some cases well beyond the expected requirements. The design tradeoff in this case is *Availability versus Performance*.

6. Immutability versus Process Functionality: This happens mainly in BC implementations supporting smart contracts. One of the key tenants of the architecture is that it is a permanent record. The inclusion of smart contracts can either violate (by having data offline or modifying previous blocks) or preserving it (by restating the smart contract, previous states and the of new state). The design tradeoff in this case is *Security versus Usability*.
7. Consensus Algorithm Selection: This tradeoff relates to the selection of the consensus approach to validation. In Nakamoto for example, the PoW constitutes both a security approach (51% of the computation) and validation of the block (encryption/finding the "nonce") while in other approaches, this could be varied to provide a level of accuracy and protection that may not be as computationally intense. The design tradeoff in this case is *Security versus Performance*.

4.3. Architecture Tradeoff Matrix

To consider the key tradeoffs for the BC pattern in the early design stage, we propose the use of a tradeoff matrix such as that shown in Table 1. The numbers in cells refer to those given in Subsection 4.2 and are scaled from zero (0) to ten (10) with ten indicating the highest tradeoff or negative correlation between a given architecture tradeoff pair and zero (0) indicating no correlation whatsoever.

Table 1. Key Architecture Quality Attribute Tradeoff Matrix

	Availability	Interoperability	Modifiability	Performance	Security	Testability	Usability	Extensibility
Availability	X			5				
Interoperability		X			2			4
Modifiability			X					
Performance	5			X	7		1	
Security		2		7	X		6	
Testability						X		
Usability				1	6		X	3
Extensibility		4					3	X

5. Quality Attribute Tradeoff Discussion

It is best practice to design new software systems with consideration of all quality attributes in the software engineering community [23]. However, our experiments indicate that one should also determine and consider key attributes identifiable by the usage of patterns that facilitate them and are embedded in design [28]. Furthermore, it is certainly the case that some quality attributes are especially dominant in certain architecture types and will typically require additional tradeoff analysis to solve [29]. This philosophy is applied to the BC design pattern here and is summarized and reflected in Figure 2. Domain knowledge of the overall design pattern, the specific use-case of the pattern, and aspects of lower-level abstractions regarding the supporting data structures and algorithms are all important factors in enabling the tradeoff matrix to be constructed.

5.1. Key AttributeTradeoff Discussion

In terms of importance and from a purely numeric perspective, it is evident that performance, security and usability are the most impacted attributes. This intuitively makes sense since performance is taxed by adding requirements for security and usability, both of which require additional resources and potentially more design impacting performance decisions. The interoperability and extensibility pair is a close second in this tradeoff analysis. This observation also

intuitively makes sense since blocks are dependent on each other to provide and deliver BC functionality in compliance with the system requirements. The variety of BC design patterns requires extensibility to enable design additional features that may be present in a future or updated system requirements specification. Finally, in terms of availability, only one tradeoff is noted. However, availability is a very important quality attribute since if the BC system becomes unavailable or underperforms it will render the system unusable.

5.2. Use of the Key Attribute Tradeoff Matrix

While all attributes should be considered in the design decisions and, in fact, most design patterns do incorporate them [30], the objectives of this investigation are to determine and concisely convey which quality attributes seem to be more important with respect to the BC pattern. We propose that during the early design phase of systems employing the BC pattern, the tradeoff matrix should be used to guide periodic evaluation activities to ensure that identified key attributes remain compliant with system requirements.

6. Evaluation of the Consensus Algorithm via Simulation

An example of how key quality attributes and their corresponding tradeoffs, as represented in a tradeoff matrix, can be considered in designing a BC solution is provided. One of the most important factors that contributes to the variability among different BC architectures is the type and characteristics of the employed consensus algorithm. This particular architectural tradeoff is designated as number 7 from the list in Subsection 7 and is observed to exhibit a key tradeoff among the *performance* and the *security* quality attributes as shown in Figure 2. In our experiment, we analyzed a simulation of two common consensus algorithms. One of those two consensus algorithms is a particular implementation from the family of Byzantine Fault Tolerance Algorithms denoted as BFTA. The other consensus algorithm is an implementation referred to as RAFT. RAFT is a consensus approach that was originally created by researchers at Stanford University [17].

6.1. Consensus Algorithm Selection and Experiement Parameters

We selected the BFTA algorithm due to its widespread usage in many BC implementations and studies; not only in BC-based systems but also in replicated database studies. We selected the RAFT algorithm because, although newer, it rapidly gaining popularity in the BC community for a variety of reasons including its simplicity.

A brief and high-level description of the BFTA algorithm is given in the following numbered list. The description is provided in terms of the original and historical use-case rather than BC consensus so that generality can be preserved and so that lower abstraction design choices are not present that could potentially complicate the description.

1. There are a number of military Generals that are leading the Byzantine Empire's army in a siege of a particular city.
2. The Generals and their commands must all attack simultaneously at an agreed upon time in order to conquer the city.
3. The Generals communicate through the use of messengers that deliver information through verbal channels only.
4. A subset and minority of the Generals are traitorous. This subset will attempt to undermine the remaining majority of honest and loyal Generals by providing false information via their messengers. Thus, the collection of Generals is classified as either "loyal" or "disloyal" with respect to their mission.
5. At some point in time, all the loyal Generals will agree on a time of attack. When the majority of all Generals, both loyal and disloyal, agree upon a common time of attack, then the attack time negotiations halt and the agreed upon time is adopted by the entire army. After adoption of an attack time, no further communication or interaction

concerning the attack time occurs. Determining when the attack time occurs is the "solution" to the problem and it is based upon achieving a "consensus" among the entire group of Generals that comprise both the loyal and disloyal disjoint subsets.

It should be clear to the reader with a basic understanding of BC, that the Generals represent BC network nodes that validate a transaction and that the disloyal Generals represent nodes that incorrectly validate (or invalidate) a transaction. The nodes represented by disloyal Generals may be undergoing an error, may have been compromised, or may be malicious actors in the BC network.

An alternative consensus methodology is embodied in the RAFT algorithm that is summarized in the following numbered bullet list. In this case, the high-level description of the algorithm is provided directly in terms of BC system components.

1. There are a number of nodes in a network that can carry out computations and participate in the operations required to commit blocks to the distributed ledger in the form of a BC.
2. Time is partitioned into subsequent discrete intervals or epochs with the duration of each epoch being chosen at random.
3. Before the next epoch in the sequence commences, a "trust election" occurs where each of the participating BC nodes votes for a node based upon some arbitrary criteria. The criteria could include characteristics such as current load, recent activity, or any of a number of potential system characteristics. The node with majority of votes wins the trust election.
4. If there is a tie then another election begins immediately. Tie avoidance can be achieved through designing the system such that only an odd number of voters are used.
5. The winning node issues the next epoch's time duration and a secret that is used by all nodes in encryption of transactions during the epoch.
6. As an epoch ends, another trust-election begins and the process continues.

While the simplicity of the RAFT approach is appealing since simplicity often leads to favorable quality attributes, it is also noted that care should be taken in the design of the trust election and voting process. One of the main benefits and motivating factors for using a BC is to avoid using a trusted centralized authority to maintain a transaction ledger since such a centralized authority becomes a critical point of failure or a highly vulnerable insertion point for an attack vector. Likewise, if the accumulation of votes and determination of the trust election winner is accomplished by a centralized voting node, similar issues can arise. This can lead to the need for a consensus algorithm within the overall BC consensus methodology. Solutions to this problem are present in the literature and the overall result of RAFT is that the winner of the trust election is accomplished in a just-in-time manner at random intervals that are unknown in the previous epoch.

6.2. Experimental Results and Conclusions

We generated software using the *C* programming language to simulate the execution of the BFTA and RAFT consensus algorithms. Furthermore, we evaluated the consensus algorithms in two different environments for running the processes. The "bare metal" environment consisted of an Apple Macintosh computer with an Intel quad-core CPU that is running the Ubunto Linux OS. The "hyper-vised" environment also used the same Ubunto Linux OS but in a virtual machine. We used the VMWare Workstation Pro 14 virtualization software for the second environment. In terms of main memory, we used identical RAM allocations of 2 GB for both environments. Additionally, we evaluated behavior as the size of the BC was varied in each environment. For this portion of the experiment, we ran simulations with BCs of sizes, 1, 2 and 3 million blocks. For each configuration and BC size, we recorded CPU usage as a percentage of overall usage and the time required to build the BC. These experimental results are provided in Table 2.

While the experimental results in Table 2 provide an interesting comparison of BFTA with RAFT, the more important point here is that such simulations can be very valuable in evaluating the resultant behavior of a consensus algorithm with respect to implementation variations before the operational BC system is developed and tested. This is an essential capability if the designer is going to utilize tools such as the key attributes tradeoff matrix disclosed here. While the tradeoff matrix points to the key tradeoff characteristics and thus where the designer should focus their attention during design, the validation of design choices made due to consideration of the matrix can be accomplished via a simulation as shown here.

It is important to underscore the fact that the simulation capability is one way to incorporate and use the tradeoff matrix in the early design stages of a BC system. There are a variety of tools that can be used by an architect in order to evolve a particular architectural design. One of these tools is simulation. In some cases, it is extremely challenging to predict how a design choice will affect a quality attribute until a model is constructed and simulated.

Table 2. Experimental Results with Native "bare-metal" and Virtual Hosts

"Bare-metal" Platform				**Virtual "hyper-vised" Host**		
Volume	***% CPU***	***Time (min:sec)***		***Volume***	***% CPU***	***Time (min:sec)***
1M Blocks				1M Blocks		
Byzantine	94%	2:37		Byzantine	95%	2:31
RAFT	97%	1:06		RAFT	87%	0:48
2M Blocks				2M Blocks		
Byzantine	93%	6:33		Byzantine	95%	5:05
RAFT	91%	2:11		RAFT	94%	1:41
3M Blocks				3M Blocks		
Byzantine	93%	9:52		Byzantine	95%	7:39
RAFT	92%	3:18		RAFT	96%	2:34

7. Conclusions

We propose the use of a tradeoff matrix that corresponds to the tradeoffs among the major software architecture quality attributes in evolving BC solutions. The tradeoff matrix is a starting point for the BC architect as key attributes are enhanced, or alternatively, de-scoped, in the design of a new BC architecture. While this approach can be generally used for any software pattern, we also focused specifically on the BC pattern and provided a tradeoff matrix based upon key quality attributes that we determined from a review of the literature as well as our own experience and experiments. We furthermore described how simulation can be used during the early architectural design phase of a BC pattern that is guided by the tradeoff matrix and whereby the simulation tool provides a basis of comparison for architectural design choices. We emphasize that simulation is not the only tool available for validation of the tradeoff behavior, but it is a good tool to consider for generating checkpoints during the architectural evolution stages of the software design process. We have shown how tradeoffs can lead to specific experiments that can be used to define and test a hypothesis for incorporation into the next generation of design during the architectural evolution process via the use of simulation. Ultimately, the system architect who is employing the BC pattern must decide on the level of detail needed to ensure the generated software solution will comply with the other necessary quality attributes to result in a feasible and resilient result; however, we believe that the identification of dominant quality attributes as provided here in combination with the use of a tradeoff matrix and architectural evolution simulations is a viable approach for employing the BC pattern in a wide variety of use-cases.

References

[1] D. Li, R. Du, Y. Fu, M. H. Au: "Meta-Key, A Secure Data-Sharing Protocol Under Blockchain-Based Decentralized Storage Architecture" *IEEE Networking Letters* 2019(1), p. 30 – 33.

[2] J.M. Medellin and M.A. Thornton, "A Discussion on Blockchain Software Quality Attribute Design and Tradeoffs" in proc. *International Conference on Emerging Topics in Computing*, August 19-20, 2019, Springer Nature, LNICST vol. 285, pp. 19 – 28, July 14, 2019.

[3] S. Nakamoto; "Bitcoin: A Peer-to-Peer Electronic Cash System", Available: www.bitcoin.org [last accessed March 19, 2019].

[4] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, K.-L. Tan, "BLOCKBENCH: A Framework for Analyzing Private Blockchains" Available: https://arxiv.org/pdf/1703.04057.pdf [last accessed March 19, 2019].

[5] A. Dorri, S. S. Kanhere, R. Jurdak, "Towards an Optimized BlockChain for IoT" in proc. *IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation* (IoTDI), pp. 173 – 178, 2017.

[6] J.M. Medellin, M.A. Thornton: "Simulating Resource Consumption in Three Blockchain Consensus Algorithms" in proc. *International Conference on Modeling, Simulation & Visualization Methods*, pp. 21 – 27, July 2018.

[7] A. Dorri, S. S. Kanhere, R. Jurdak, P. Gauravaram, "Blockchain for IoT security and privacy: The case study of a smart home" in proc. *IEEE International Conference on Pervasive Computing and Communications Workshops* (PerCom), pp. 618 – 623, 2017.

[8] T. Salman, M. Zolanvari, A. Erbad, R. Jain, M. Samaka, "Security Services Using Blockchains: A State of the Art Survey" *IEEE Communications Surveys & Tutorials* 21(1), pp. 858 – 880.

[9] J.M. Medellin, M.A. Thornton, "Performance Characteristics of Two Blockchain Consensus Algorithms in a VMWare Hypervisor" in proc. *International Conference on Grid & Cloud Computing and Applications*, pp. 10 – 17, July 2018.

[10] L. Lamport, R. Shostak, M. Pease, "The Byzantine Generals Problem" *ACM Transactions on Programming Languages and Systems* 4(3), pp. 382-401.

[11] Q. Xia, E. B. Sifah, K. O. Asamoah, J. Gao, X. Du, M. Guizani: "MeDShare, "Trust-Less Medical Data Sharing Among Cloud Service Providers via Blockchain" *IEEE Access* 5(1), pp. 14757 – 14767.

[12] S. Muralidharan, C. Murthy, B. Nguyen et al., "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains", Available: https://arxiv.org/pdf/1801.10228.pdf [last accessed March 19, 2019].

[13] X. Liang, T. Wu, "Exploration and Practice of Inter-bank Application Based on Blockchain" in proc. *12th International Conference on Computer Science & Education* (ICCSE) pp. 219-224, 2017.

[14] K. Christidis, M. Devetsikiotis, "Blockchains and Smart Contracts for the Internet of Things" *IEEE Access* 4(1), pp. 2292 – 2303.

[15] F. Daniel, L. Guida, "A Service-Oriented Perspective on Blockchain Smart Contracts," *IEEE Internet Computing* 23(1), pp. 46 – 53.

[16] W. Stallings, Cryptography and Network Security, Principles and Practice, 7th edition, Pearson Education Limited, London, United Kingdom, 2018.

[17] N. Ferguson, B. Schneier, T. Kohno, Cryptography Engineering, Design Principles and Practical Applications, Wiley Publishing, Inc., Indianapolis, Indiana, 2010.

[18] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm" in proc. *USENIX Annual Technical Conference*, pp. 305-319, 2014.

[19] D. Fullmer, A. S. Morse, "Analysis of Difficulty Control in Bitcoin and Proof-of-Work Blockchains" in proc. *IEEE Conference on Decision and Control* (CDC), pp. 5988 – 5992, 2018.

[20] J. Golosova, A. Romanovs, "The Advantages and Disadvantages of the Blockchain Technology" in proc. *IEEE 6th Workshop on Advances in Information, Electronic and Electrical Engineering* (AIEEE), pp. 1 – 6, 2018.

[21] C. Ehmke, F. Wessling, C. M. Friedrich, "Proof of Property – A Lightweight and Scalable Blockchain Protocol" in proc. *IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain* (WETSEB), pp. 48 – 51, 2018.

[22] E. Ceccetti et al., "Solidus: Confidential Distributed Ledger Transactions via PVORM" in proc. CCS, pp. 1 – 23, 2017.

[23] Chao-Yang Li, Xiu-Bo Chen, Yu-Ling Chen, Yan-Yan Hou, Jian Li, "A New Lattice-Based Signature Scheme in Post-Quantum Blockchain Network" *IEEE Access* 7(1), pp. 2026 – 2033.

[24] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, The SEI Series in Software Engineering, 3rd edition, Addison Wesley, Upper Saddle River, New Jersey, 2012.

[25] H. Cervantes, R. Kazman, Designing Software Architectures; A Practical Approach, The SEI Series in Software Engineering, Pearson Education, Boston, Massachusetts, 2016.

[26] B. A. Scriber, "A Framework for Determining Blockchain Applicability" *IEEE Software* 35(4), pp. 70 – 77.

[27] X. Xu, I. Weber, M. Staples, L. Zhu et al., "A Taxonomy of Blockchain-Based Systems for Architecture Design" in proc. *IEEE International Conference on Software Architecture*, pp. 243 – 252, 2017.

[28] Z. Zheng, S. Xie, H. Dai, X, Chen, H. Wang, "An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends" in proc. *IEEE International Congress on Big Data* (BigData), pp. 557 – 564, 2017.

[29] G. Booch, Object-Oriented Analysis and Design with Applications, Addison Wesley Longman, Inc., Reading, Massachusetts, 1994.

[30] J. Tian, Software Quality Engineering; Testing, Quality Assurance and Quantifiable Improvement, Wiley, Hoboken, New Jersey, 2005.

[31] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd edition, Pearson Education, Inc., Upper River, New Jersey, 2005.