Smart-Contract Execution with Concurrent Block Building

Lian Yu
School of Software & Microelectronics
Peking University
Beijing, 102600, P.R. China
lianyu@ss.pku.edu.cn

Guannan Li, Yafe Yao, Chenjian Hu
School of Software & Microelectronics
Peking University
Beijing, 102600, P.R. China
guannan.li@ pku.edu.cn

Wei-Tek Tsai
School of Computer Science & Engineering
Beihang University, Beijing, China
Comp., Info., and Dec. Systems Engineering
Arizona State University
Tempe, AZ 85287, USA

Enyan Deng
Beijing Tiande Tech
Beijing, 100089, P.R. China
deng@tiandetech.com

Abstract— **Business processes are often related to operational processes, contracts, and regulations. Modeling such processes needs to address regulation monitoring and enforcement, and maintain a reliable history of data for evidence. This paper proposes modeling business processes as smart contracts (SCs) on permissioned blockchains (BCs). The challenges with the proposed approach are state synchronizations among distributed nodes (called authnodes), and real-time requirements. This paper separates the executions of SCs from the state managements on multi-BCs, and proposes a pipeline model to verify and create blocks in parallel.**

Keywords— blockchains, smart contracts; permissioned blockchain; concurrent block building.

I. INTRODUCTION

Blockchains (BCs) and smart contracts (SCs) have received significant attention recently where an SC is an executable code running on top of a BC. SCs are a collection of computer protocols that facilitate, verify, or enforce the negotiation or performance of a contract by automatically executing the terms of a contract, reducing transaction costs associated with contracting, and hopefully providing assurance better than conventional paper-based contract management.

The SC was proposed by Nick Szabo on digital contracts and digital currency [1][2]. This concept implies that executable code will become a legal contract that can be executed, and the results produced can be legally binding. These SC concepts were not realized then because no financial systems can support programmable transactions.

Later, this concept came back when Bitcoins and other digital currencies became popular. But the SC concept has changed slightly, and this time an SC runs on top of a BC that supports a digital currency. For example, Ethereum [3], an open-source project, supports SCs. In Ethereum, there are two types of entities that can generate and receive transactions: people and contracts. A contract is essentially an automated agent that lives on the Ethereum BC, has an address and a balance, and can send and receive transactions. The legal aspect is not emphasized, but programmability and execution are priority items.

Financial institutions often have complex business processes with many participants, these systems need to address these issues not necessarily needed for cryptocurrency:

(1) They need to control access permissions, such as *read* from or *write* into BCs while maintaining BCs properties. This needs to be true regardless if it is a public or private BC.

(2) They have complex business processes with multiple parties participating. Before an asset (money, stocks, bonds or others) can be placed into a SC for execution, its ownership needs to be cleared. Then the SC should process the transaction legally, and store the results into BCs.

This paper proposes to implement business processes as SCs on permissioned BCs, i.e., only those nodes that have permissions can verify and vote in the block-creation process in this kind of BCs. Permissionless BCs are those BCs where any node can vote/participate in the block creation process. This approach has the following features:

(1) There is a set of distributed authentication nodes called authnodes that communicate through a high-speed network instead of a P2P (peer-to-peer) network, and they synchronize with each other;

(2) It has a voting mechanism to keep every node consistent;

(3) It separates account BCs to store process states in-progress and the results;

(4) It has a recovery mechanism to roll back in case of failures.

978-1-5090-6320-8/17 $31.00 © 2017 IEEE
DOI 10.1109/SOSE.2017.33

This paper is organized as follows: Section II presents the related work; Section III describes the proposed architecture; Section IV introduces a pipeline model of block creation and presents synchronizations among authnodes; Section V designs a permissioned BC to work with SCs, and Section VI concludes this paper.

II. Related Work

This section briefly presents BCs, SCs and business related work.

A. BCs and SCs

Numerous BCs have been designed including those included in cryptocurrencies such as Bitcoin and other applications. A BC can be a permissionless BC where everyone can be involved in block creation, or a permissioned BC where only selected nodes can be involved in block creation. In [13, 14], three new BCs have been proposed:

- TBC (Trading BC): A TBC will involve in trading among nodes, but it does not keep track account values unless those account values are needed for trading. After it performs trading, it will send the updated value to relevant ABCs.
- ABC (Account BC): An ABC will involve in bookkeeping accounts, but it will not be involved in trading. An ABC will supply account values to a TBC for trading.
- MBC (Message BC): An MBC will store messages communicated in the system. It is used to ensure that the system operations can be reconstructed if necessary as all the communications among parties are recorded. This feature is important for financial systems as well as military systems.

These new BC designs allow scalability and address privacy issues commonly encountered in existing BC designs.

Originally, SCs are defined as computer protocols that execute the process specified in a legal contract [10]. As most of legal contracts are paper based, SCs are superior to paper-based contracts they are executable and thus can complete the process specified in a contract in an automatic manner. The code, also like traditional contracts, defines things should be done by different parties that signed the contract, such as duties, interests and penalties specified in the contract.

Recently, the meaning of SCs has changed. Instead of being an executable legal contract, SCs are the code segments that can be executed on a BC, as the BC may store digital assets (such as money or stocks), these SCs control those digital assets. The inputs to the SC must be stored in the BC before execution, and the execution results will be stored in the BC.

For example, SCs in Ethereum, a public BC, are a collection of code (its functions) and data (its states) that reside at specific addresses on the BC. Contracts stay on the BC in a binary format called Ethereum Virtual Machine (EVM) and it will execute bytecode. SCs are written in high-level languages such as Solidity, Serpent, and LLL, and then compiled into bytecode to be uploaded on the BC [10].

The EVM bytecode consists of a series of bytes, and each byte represents an operation. An index pointer is maintained, the index will increase by one after every operation, and continuously processing the operation found at the current index pointer until the code execution is finished or a failure occurs. Each operation requires a fee and the balance on the account (the sender of the transaction) will be reduced [12].

An Ethereum SC is "activated" every time someone sends a transaction to it, at which point it runs its code, perhaps modifying its internal state or activating other SCs, and then shuts down. The states are organized as Merkle Patricia tree, and the root of the tree is stored in the blockhead along with the transactions list tree (i.e., Merkle tree) root. Each "state tree" represents the current state of the entire system, including address account balances and contract states [3].

Each node of Ethereum processes every transaction on its list and changes its state until a block is built successfully. Then the state tree root will be written in the block header and the block is ready to be sent out (if a transaction is SC related, it should wait for the contract to complete so that the state can be confirmed). When a node receives a block, it processes every transaction in the block onto the parent block's state and calculates new state, then checks if the new state root matches state root in block header [3]. If matching, the block will be accepted.

This SC processing approach may not be suitable for those contracts that have a long running time because of the following reasons:

- **High execution cost.** Contracts are written in high-level language and then compiled into bytecode to be executed, and each operation requires a fee, thus those contracts with many operations will incur a high fee.
- **Slow down the block-building processing.** When a node processes transactions that will activate SCs, it will activate and execute these SCs one after another. This will slow down the transaction processing and the block-building processing.
- **Slow down the transaction confirmation processing.** Once a transaction is included in a block, it means that the transaction has been processed by a majority of nodes. But like building a new block, every node verifies a block it receives, and needs to process transactions in the block one after another. This costs significant time for those contracts.

To enhance the capabilities, SCs need to have business process modeling support and run on efficient BCs.

B. Business Process and Modeling

Business Process Model and Notation (BPMN) is a graphical representation for specifying business processes in a business process model [11], as a standard proposed and maintained by Object Management Group (OMG) that formally released BPMN 2.0 version in January 2011, and the execution semantics were also introduced alongside the notational and diagramming elements. BPMN defines a set of steps described in the business process of the graphic symbol, and describes end-to-end business processes.

The graphic symbols are specifically designed to coordinate the process in a group of related collection of activities, and delivering messages between different participants in the process. BPMN 2.0 model can perform in any engine compatible with the standard, and can also exchange between graphics editor. BPMN makes enterprises have the ability to understand their internal business processes through graphic symbols, and discuss their business process on a unified standard. In addition, the graphic symbol can improve the efficiency of the mutual cooperation between enterprises, and help enterprises understand businesses between themselves and their partners, so that enterprises can quickly adapt to the new internal and B2B business scenarios.

As a case study, this paper re-engineers a BPMN 2 process engine, Activiti (http://activiti.org/), such that business processes can be implemented as SCs that can have powerful process-modeling capabilities, and at the same time, maintaining those BC properties.

III. Proposed Architecture

Figure 1 shows the system architecture with five layers from the bottom to the top: Caching, BC Services, APIs, SCs, and Applications Layers.

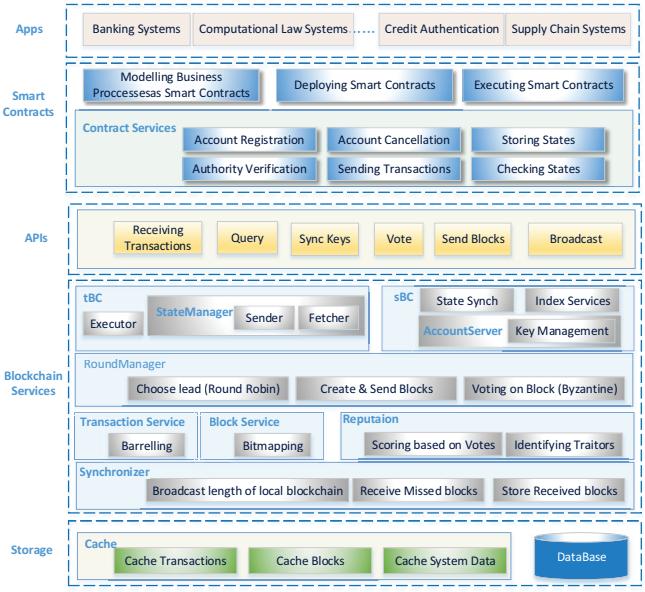

Fig. 1. System Architecture

- **Caching Layer:** This is to cache temporary information in the memory, including new transactions received from users and SCs; those blocks not yet transferred to disk; and temporary data that support the system running.
- **Blockchain-Services Layer**: Transaction service puts transactions in the cache into barrels; Block service creates a bitmap for the transactions in each barrel; RoundManager chooses a leader with a round-robin way, creates and sends a block to all other authnodes, further performs reputation computing; Synchronizer broadcasts the length of local BC, receives the missed blocks, and stores the received blocks; for SC transactions, TBC (Trading BC) executes the SC code first, and then puts the results into barrels, puts into barrels directly for non-SC transactions, and is ready for creating blocks; SBC (State BC) synchronizes BCs to ensure consistent state of different authnodes, creates account index to accelerate query, and provides account public-private key services.
- **APIs Layer**: This provides interfaces external and internal APIs. The internal APIs are used for internal communication between authnodes, such as voting, broadcasting blocks. External APIs are used for external users, such as accepting the new transactions and the query operations.
- **SC Layer**: This provides contract-related services. SCs are written according to the domain-specific requirements, lawfully and rightly verified by all stakeholders, and then deployed in BC system to execute. This layer has three functions: interaction (editor) with the users, process execution engine and contract services that support account management, state storage, and sending transactions.
- **Application Layer**: This layer runs applications, such as bank systems, computational law systems, credit certification systems, and supply chain systems.

IV. Creating SCs for Business Processes

This section discusses issues related to creating SCs, and the methodology to handle these issues. Organizations need to share information among different units, integrate functions, and exchange a variety of goods and information. Many business processes are automated such as sending documents, exchanging data, executing tasks among participants based on pre-defined rules. Compared with contracts, business processes are often more complicated, and require a long life cycle, e.g., a trade-finance process may take months to complete.

A. Business Processes as SCs

When implementing business processes as SCs on BCs, one will face new issues as follows:

(1) Many processes will need real-time response, thus timing constraints, scheduling, fault-tolerance, and scalability are important;

(2) Different participants will need different BCs with different functionality and performance;

(3) Participants need to synchronize with each other;

(4) SCs need to be lawfully and formally verified; and

(5) SCs need to recover in case of system failure during execution.

Among them, state synchronizations among participants (i.e., authnodes) are critical. This paper defines the system states into three types: account states, local states and federal states, and they form a hierarchical structure as shown in Figure 2.

- Account states: This stores any application or domain information such as bank accounts and product inventory. An account may have several fields, and any changes of

these fields will lead to the change of the account state at that authnode. An account state is the smallest state unit in the system, and may be changed due to an execution of a SC.

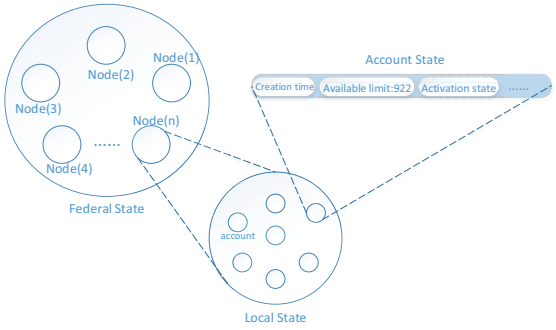

Fig. 2. Three Types of States at Different Levels

- Local states: A SC system consists of multiple distributed authnodes. Each authnode may have several types of accounts, and aggregating the states of account states will form a local state. Any changes of the account states in the node lead to changes of the local state.
- Federal states: The states of all authnodes constitute the federal states, and any changes of local states in the system will result in the change of the federal states. Federal state allows that local states of authnodes are different in a short period of time, i.e., executions of authnodes may execute SCs successively, called the intermediate states, but after the state synchronizations, these authnodes should come up with an agreement and reach a consistent state in the end.

Figure 3 shows that the federal state of the system switches between the consistent state and the intermediate state. A consistent state is a stable state in which each authnode has the same state, and if the system does not receive any new transaction, the system maintains the state unchanged. An intermediate state is a temporary state of the system. At runtime, local authnodes may receive new transactions at different times and take different computation times to complete a SC, and thus not all the nodes will have the same states during the transition period. Thus, a system may move from a consistent state to an intermediate state, and to move from an intermediate state to a consistent state, synchronization will be needed.

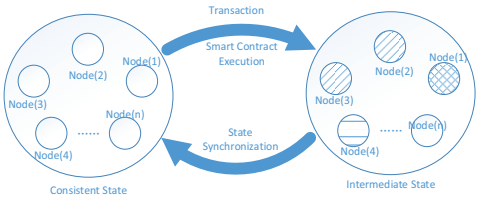

Fig. 3. Consistent State and Intermediate State

B. Sequential Execution of SCs

This model adopts the approach of SC execution in Ethereum, i.e., state synchronization in a sequential order, where a local state of a node changes first, then the local states of the other authnodes change in the same way by synchronization as shown in Figure 4.

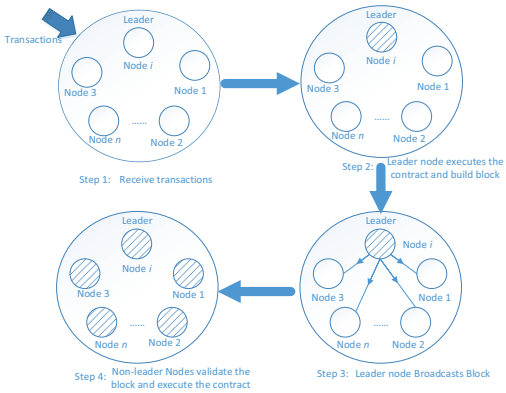

Fig. 4. Sequential Executions of SCs

Assume that the SC system is in a consistent state as shown in Figure 3, and authnode(*i*) is selected as the leader using an election algorithm to build blocks. As shown in Figure 4, the system first receives a transaction in step 1; the leader checks each transaction in the transaction set to determine whether it needs to trigger a SC in step 2, if yes, it executes the SC, and this causes the changes of local states of the leader, while the rest of local states remain unchanged; the leader broadcasts the created block to the rest of nodes in step 3, and at this moment, the system is in an intermediate federal state; finally, in step 4, each of the rest of nodes receives and verifies the block, and executes the SC triggered by transactions in the block, eventually all the nodes have the same local states.

The Ethereum contracts tend to be simple, and the execution time is short, so this model is suitable when the execution time of the contract is much shorter than the time used for building a block.

The algorithm of this process is shown in Table I, where *M* is the transaction set to be processed, *SMC* is the SC type, *B* is the block that has been built, *C* is the SC object, *Stx* is the state after the contract executes, *S* is the state set, *verifyTx*(*tx*) is a function to verify the validity of transaction *tx*, *Type*(*tx*) is to check the type of *tx*, *loadSmartContract*(*tx*) is to load the SC that *tx* specifies, *execute*(*C*) is to execute the contract *C*, *broadcast*(*B*) is to broadcast block *B*, and *buildBlock*(*M, S*) is to build block using transaction set *M* and state set *S*.

TABLE I. CREATING BLOCKS BY LEADER NODE

Algorithm: Building a block, running in leader node	
Input: transactions tx	
1	M ← transactions
2	**forall the** tx **in** M **do**
3	**if** verifyTx(tx) = false **then**
4	**return** false
5	**else**
6	**if** Type(tx) = SMC **then**
7	C ← loadSmartContract(tx)
8	Stx ← execute(C)
9	S ← Stx
10	B ← buildBlock(M, S)
11	broadcast(B)

At this point, the leader node's work has been completed and the rest of nodes will continue the follow-up task. Table II shows the pseudo-code, where *S'* is the state set in the received block, and *getState*(*B*) is to get the state from the block *B*.

TABLE II. VALIDATE A BLOCK IN SEQUENTIAL EXECUTION MODEL

```
Algorithm: Validating a block, running in other nodes
Input: a block B
1   if verifyBlock(B) = false then
2     return false
3   else
4     forall the tx in B
5     M ← getTx(B)
6     forall the tx in M
7       if verifyTx(tx) = false then
8         return false
9       else
10        if Type(tx) = SMC then
11          C ← loadSmartContract(tx)
12          Stx ← execute(C)
13          S ← Stx
14    S' ← getState(B)
15    if S' = S then
16      return true
17    else
18      return false
```

C. Parallel Execution of SCs

All the authnodes in the SC system receive a transaction at the same time, and can check the transaction type in each node when receiving it in step 1 as shown in Figure 5. If it is a contract, each authnode begins to load and execute the contract, and local states on every authnode change at the same time in step 2. By the time when non-leader nodes receive the block from the leader, they have already finished executing the contract and changed their local states in step 3. To verify and synchronize the local states, authnodes compare their local states with the state received in the block, the verification passes if the states are the same in step 4. In this model, the leader authnode and non-leader authnodes execute SCs concurrently, the building block process by the leader node is the same as the previous one, and the working process of non-leader node is shown in Table III.

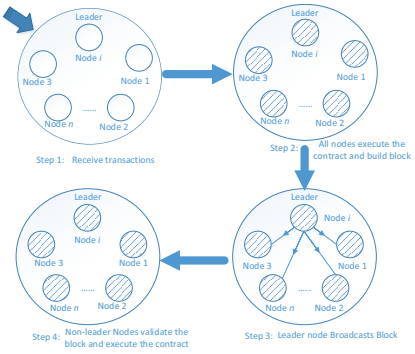

Fig. 5. Parallel Executions of SCs

TABLE III. VALIDATE A BLOCK IN SEQUENTIAL EXECUTION MODEL

```
Algorithm: Validating a block, running in other nodes
Input: a block B and local state S
1   if verifyBlock(B) = false then
2     return false
3   S' ← getState(B)
4   if S' = S then
5     return true
6   Else
7     return false
```

D. Non-blocking Execution of SCs

In the sequential execution model and concurrent execution model, the system obstructs the operations of building a block till the completion of SC executions. But in this model, authnodes decouple the process of building blocks with contract execution: when scanning a transaction and executing a SC, an authnode no longer needs to wait for the contract execution to complete as the previous two models do, it will continue to scan and execute the next transaction immediately instead.

Table IV shows the pseudo-code, where *Trigger*(*C*) invokes the SC to execute asynchronously, i.e., when the contract execution has been triggered, the transaction continues its way for building blocks, *buildBlock*(*M*) is to create blocks using transaction set *M*, and the state will not be contained in the block, because when the block has been built successfully, the SCs that were triggered by transactions contained in the block are still running, so the state is also changing simultaneously.

TABLE IV. BUILDING A BLOCK IN LEADER NODE IN NON-BLOCKING EXECUTION MODEL

```
Algorithm: Building a block, running in leader node
Input: transactions tx
1   M ← transactions
2   forall the tx in M do
3     if verifyTx(tx) = false then
4       return false
5     else
6       if Type(tx) = SMC then
7         C ← loadSmartContract(tx)
8         Trigger(C)
9   B ← buildBlock(M)
10  broadcast(B)
```

E. Anlaysis on Three SC Execution Models

In the above three models, blocks are built by the leader node. In the sequential execution model, the leader node executes SCs first, then the rest of the authnodes execute the SCs. In the parallel execution model, all the authnodes execute SCs at the same time, but like the sequential execution model, authnodes scan transactions one by one to check the transaction types, such that the executions of SCs are in the same order. The second model significantly shortens the time of verifying a block as shown in Figure 6.

In the non-blocking execution model, the contract execution process does not deter the system from building blocks, and this makes the block-building process faster than those of other two models. In this model, the account state may have been modified by different contracts at the same time, i.e., it is a critical resource, and thus the account state needs to be locked when a

contract performs *write* operations. The sequential model has the longest execution time.

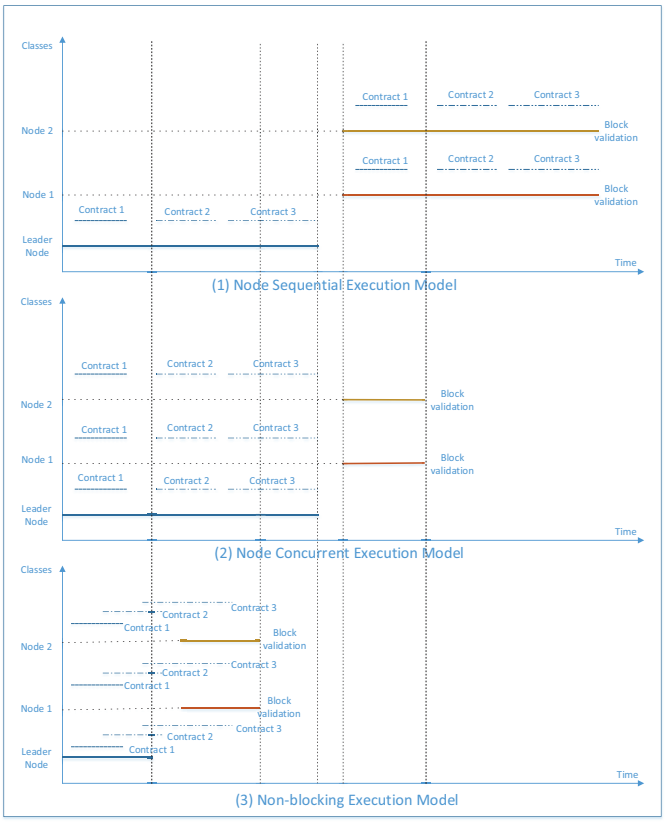

Fig. 6. Time of Block Building and Contract Executions in Three Models

F. Long-Lasting Contracts

For those long-lasting contracts, they can be divided into multiple stages, e.g., a SC can be divided into three stages *a*, *b*, c, respectively. Each stage may involve one or more state changes. The separation of the contract execution and the block building process makes the process complicated. As all the authnodes are running autonomously, and this increases the difficulty in synchronization. In the real situation, the environments of different authnodes may be different as well, thus the speeds of SC executions may differ.

Assume when the stage *b* of authnode(*i*) has been completed, authnode(*i*+1) has just finished stage *a*, obviously authnode(*i*+1) has fallen behind, and the local state cannot be directly synchronized between these two authnodes at this point, it needs to wait for authnode(*i*+1) to finish stage *b*, then the local state of authnode(*i*) and authnode(*i*+1) need to be compared and synchronized, thus the local state should be recorded every time when it changes during SC execution.

V. DESIGNING BLOCKCHAINS WITH SCS

A. Building Blocks Concurrently

The existing block-building process needs to go through four steps to decide the content of next block:

(1) Each authnode maps every transaction received by onto a bitmap; chooses a leader node in the system; and sends its own bitmap (representing those transactions that may be included in the next block) to every other authnode;

(2) Each authnode receives the bitmaps from other authnodes and determines those transactions to be included in the next block by performing intersection on the bitmaps received. The leader node creates a candidate block and sends to all other authnodes;

(3) All non-leader authnodes verify the candidate block received from the leader and send the verification results to every other authnode;

(4) All authnodes forward their voting results. After obtaining all the votes, the next block is finalized. BC system begins the next round of block building.

When the second step is completed, the system has already identified all the transactions to be included in the next block. Those transactions that did not make into the next block as well as any new transactions will be considered in the block after the next.

This system can create blocks in a pipelining manner as shown in Figure 7:

- At the beginning of building a block, assign the block a serial number height. Buffer all the transactions in a barrel, and label the barrel and all the information to be transferred such as bitmap and block with the height in step 1.1 within step 1 in Figure 7.
- In the conventional approach, the next round of block-building will begin only after the current round is completed, i.e., after the fourth step. In the pipeline model, once all the transactions are transferred from the buffer to the barrel, the system begins the next round of block building. In this way, block building will be done in a concurrent manner.
- After the end of each block building, the transactions in the buffer will include those transactions arrived after the transfer from the buffer to the previous barrel and those transactions that were not included in the previous block due to lack of votes. These transactions will be included in the new barrel.
- Except for the first block, after the completion of each block, it needs to store the hash of the previous block in the current block. So after the completion of a block, its previous block may have not completed yet, the current block will stay in the memory until the previous block has completed its process.

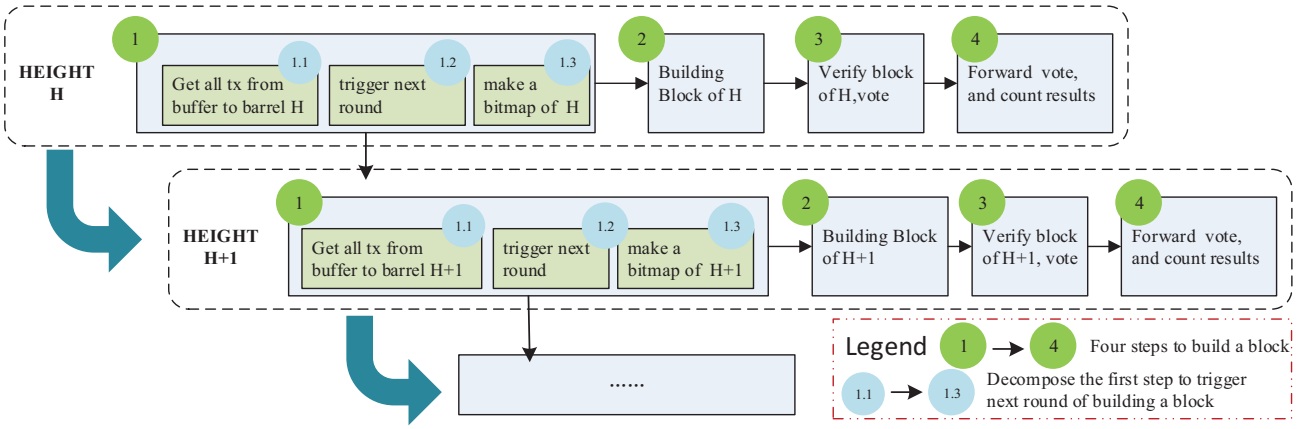

Fig. 7. Pipeline Model to Build Blocks Concurrently

B. SCs-based on Account Blockchain

The three SC execution models were presented without discussing the interaction with block building, especially with the concurrent block-building process. Furthermore, SC execution also involves with state (e.g., account, local, and federal) maintenance. SC execution, block-building and state maintenance interact with each other, designing a BC that can handle all three features at the same time will be challenging.

One possible way is to separate block-building from state maintenance, and make these two processes asynchronous. A new BC is proposed called SBC (for state BC) to manage state maintenance including state synchronization and storage. Thus, in addition to TBC, ABC, and MBC, SBC is available. SBC uses the same consensus algorithm to ensure that all authnodes contain consistent data.

Furthermore, if an account has associated active SCs, it will be placed in a TBC. A TBC is a BC that handles transactions, as a SC can trigger transactions, thus the account will be moved to a TBC ready to engage in transactions.

During SC execution, if any operation incurs any state change in an authnode, it sends a transaction (sTx for short) to the SBC to report the state change, and the SBC will be responsible to synchronize the state change among authnodes.

The interaction between TBC and SBC includes three sub-processes as shown in Figure 8: *create-account*, *execute-contract* and *state-synchronization*. It works as follows:

(1) Receive transactions from authnodes;

(2) Identify the type of transactions received;

(3) If the type is "create", then start a *create account* sub-process;

(4) If the type is not related to SCs, then transactions are stored into the TBC directly. Transactions triggering *create-account* or *execute-contracts* are also stored into TBC;

(5) If the type is "execute", then start an *execute-contracts* process; and

(6) All the state changes are synchronized by *state-synchronization* process.

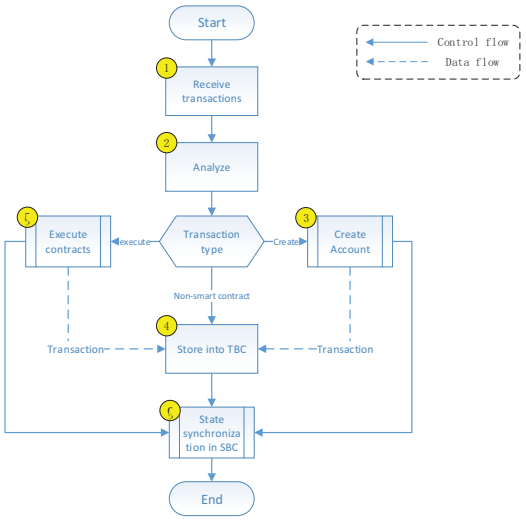

Fig. 8. Interaction Process between TBC and SBC

1) *create-account sub-process*

If it is to create a contract account, there are two steps: (1) upload the contract code and generate a "create" transaction; (2) generate an account address. If it is to create a non-contract account, i.e., those accounts without SCs, the first step is not necessary.

2) *execute-contracts sub-process*

This has three steps: (1) load the contract code based on the contract account address from the SBC; (2) fetch the initial state from the SBC, including the state of contract account and the states of execution-dependent accounts; and (3) execute the contract. Thus, by the time it starts, an authnode has already known the contract code that it needs to execute.

3) *state-synchronization sub-process*

This process has the following steps as shown in Figure 9:

(1) Check the state transactions and determine which account's state the transactions will update. These transactions are sent from the TBC;

(2) Generate stage barrels using information in state transactions, such as the contract account address and the number of times this contract has been triggered

since last synchronization. And then start a new timer for each barrel;

(3) Calculate state results in the barrel received from authnodes, divide into different groups such that each group has the same results, and label the groups with the numbers of the results inside;

(4) Check the results of step (3) to see if any group in a barrel has reached a consensus (more than 2/3 authnodes in the TBC have the same state). If yes, go to step (5), if not, step (7);

(5) Store the state into the SBC;

(6) Record the barrel as state consistent, then delete the barrel;

(7) Check the barrel timer to see if it has timed out, if it has, go to step (3), otherwise go to step (8);

(8) Record the barrel as timed out, then delete the barrel.

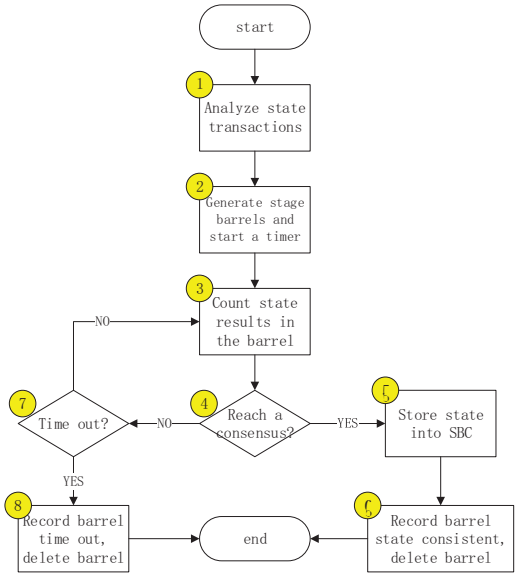

Fig. 9. *State-Synchronization* Sub-Process

One still needs to address the case with long-lasting SCs with multiple stages as discussed in Section IV (F). For the same problem where stage *b* on authnode(*i*) has completed, but authnode(*i*+1) has just finished stage *a*, earlier solutions will require involved processes to wait.

But with SBC, an authnode will send the sTx to the SBC if it involves any state change at any execution stage. In this case, authnode(*i*) has sent the sTx(*a*), sTx(*b*), and the SBC caches the sTxs, and waits for the rest of authnodes. At that moment, authnode(*i*+1) has sent the sTx(*a*), so the SBC needs to vote to confirm if the state of authnode(*i*) and authnode(*i*+1) has changed in the same way, and this is done by comparing the two sTx(*a*) received from the two authnodes. After reaching an agreement, sTx(*a*) is written into the SBC. The SBC will continue to wait for sTx(*b*) of authnode(*i*+1) when it has waited for more than a certain period of time and still no sign of receiving sTx(*b*) of authnode(*i*+1), it will not wait any longer, thus sTx(*b*) will not be written into the SBC, namely sTx(*b*) has not reached an agreement.

In addition, the SBC dynamically maintains a recent state index for each account so that the latest status can be rapidly retrieved.

VI. Conclusion

This paper analyzes existing SC techniques, and proposes a process-oriented SC technique. The paper explores SC execution models including sequential, parallel and non-blocking execution models. Furthermore, this paper proposes a pipeline model to build blocks and a new BC design SBC to facilitate state synchronization. These problems stated are new and the solutions proposed are original.

References

[1] Nick Szabo. "Formalizing and Securing Relationships on Public Networks, First Monday

[2] Nick Szabo. “A Formal Language for Analyzing Contracts”, http://szabo.best.vwh.net/contractlanguage.html

[3] “Ethereum: A Next-Generation Generalized Smart Contract and Decentralized Application Platform" http://ethereum.org/ethereum.html

[4] J. H. Fowler, T. R. Johnson, J. F. Spriggs, S. Jeon, and P. J. Wahlbeck. "Network Analysis and the Law: Measuring the Legal Importance of Precedents at the U.S. Supreme Court." Political Analysis 15.3 (2006): 324-46.

[5] Ben Shneiderman and Aleks Aris. "Network Visualization by Semantic Substrates." IEEE Transactions on Visualization and Computer Graphics 12.5 (2006): 733-40.

[6] Michael J. Bommarito and Daniel M. Katz. "A Mathematical Approach to the Study of the United States Code." Physica A: Statistical Mechanics and Its Applications 389.19 (2010): 4195-200.

[7] Michael J. Bommarito, Daniel Martin Katz, Jonathan L. Zelner, and James H. Fowler. "Distance Measures for Dynamic Citation Networks." Physica A: Statistical Mechanics and Its Applications 389.19 (2010): 4201-208.

[8] James H. Fowler and Sangick Jeon. "The Authority of Supreme Court Precedent." Social Networks 30.1 (2008): 16-30.

[9] Michael J. Bommarito, "Empirical Survey of the Population of US Tax Court Written Decisions, An." Va. Tax Rev. 30 (2010): 523.

[10] What Are Smart Contracts? Cryptocurrency's Killer App http://www.fastcompany.com/3035723/app-economy/smart-contracts-could-be-cryptocurrencys-killer-app

[11] Business Process Model and Notation https://en.wikipedia.org/wiki/Business_Process_Model_and_Notation

[12] Ethereum Homestead--Create and deploy a contract http://ethdocs.org/en/latest/contracts-and-transactions/contracts.html#create-and-deploy-a-contract

[13] W. T. Tsai, R. Blower, Y. Zhu, and L. Yu, “A System View of Financial Blockchains,” Proc. of IEEE Service-Oriented System Engineering, 2016.

[14] W. T. Tsai, L. Feng, H. Zhang, Y. You, L. Wang, and Y. Zhong, “Intellectual-Property Blockchain-based Protection Model for Microfilm,” Proc. of IEEE Workshop on Blockchains and Smart Contracts, 2016.