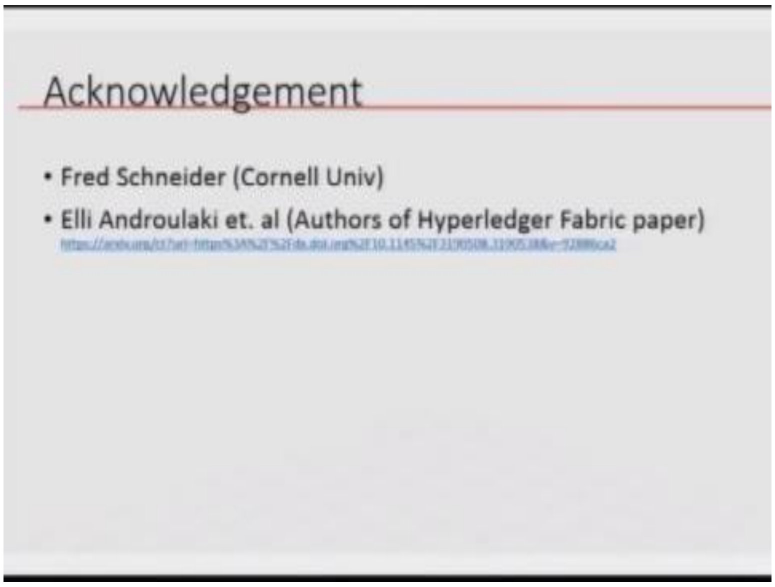
Introduction to Blockchain Technology & Applications

Prof. Sandeep Shukla
Department of Computer Science and Engineering
Indian Institute of Technology-Kanpur

Lecture - 21

Welcome to another session of blockchain technology and applications. So last time, we talked about the superficial overview of how Hyperledger blockchain works. What we want to do in this session and the next session is to actually take you a little bit inside the theory that is behind the particular design choice that the Hyperledger people made. And if you understand these design choices, you will understand the goal of Hyperledger blockchain or similar enterprise blockchains.

(Refer Slide Time: 00:50)

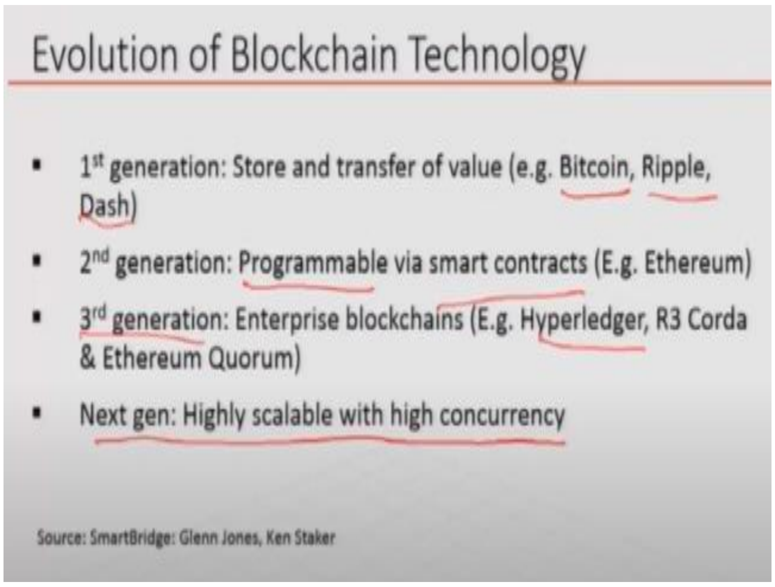


Acknowledgement

- Fred Schneider (Cornell Univ)
- Elli Androulaki et. al (Authors of Hyperledger Fabric paper)
  [illegible]

So much of this material is from Fred Schneider and the original Hyperledger Fabric paper that is available in archive.

(Refer Slide Time: 00:59)



Evolution of Blockchain Technology

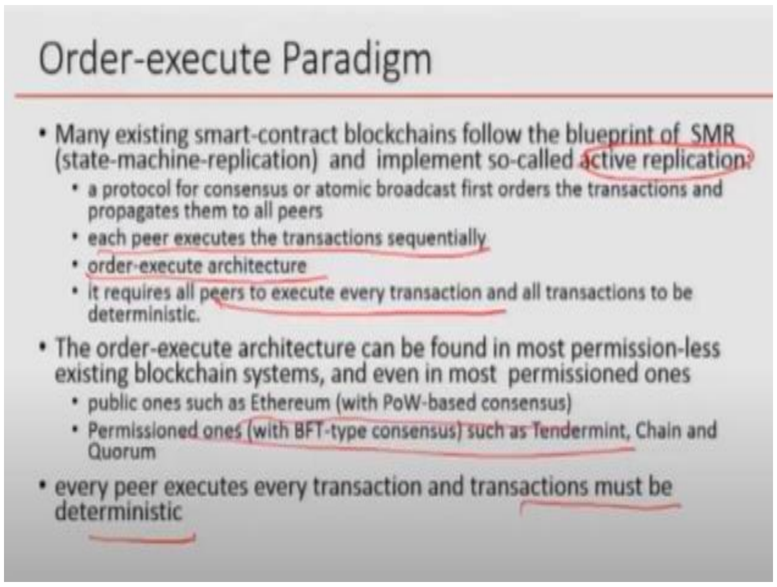
- 1st generation: Store and transfer of value (e.g. Bitcoin, Ripple, Dash)
- 2nd generation: Programmable via smart contracts (E.g. Ethereum)
- 3rd generation: Enterprise blockchains (E.g. Hyperledger, R3 Corda & Ethereum Quorum)
- Next gen: Highly scalable with high concurrency

Source: SmartBridge: Glenn Jones, Ken Staker

So we have been discussing this several times, but there is no harm in repeating. That the first generation of blockchain which basically means bitcoin and similar cryptocurrency, blockchains, like Ripple coin, Dash coin etc., is basically for storing and transferring of value. They had very limited ability to program the transactions. And that limitation kind of prompted the idea of Ethereum and similar second generation blockchain, which is programmable via smart contracts.

We are now seeing is what we call third generation blockchain and or enterprise blockchain. And enterprise blockchain, one of the early forerunners of the enterprise blockchain is Hyperledger. And then we will also see Corda very soon. And then soon there will be other blockchains, which are highly scalable with high concurrency. But let us focus now on Hyperledger.

(Refer Slide Time: 02:08)



Order-execute Paradigm

- Many existing smart-contract blockchains follow the blueprint of SMR (state-machine-replication) and implement so-called active replication:
  - a protocol for consensus or atomic broadcast first orders the transactions and propagates them to all peers
  - each peer executes the transactions sequentially
  - order-execute architecture
  - it requires all peers to execute every transaction and all transactions to be deterministic.
- The order-execute architecture can be found in most permission-less existing blockchain systems, and even in most permissioned ones
  - public ones such as Ethereum (with PoW-based consensus)
  - Permissioned ones (with BFT-type consensus) such as Tendermint, Chain and Quorum
- every peer executes every transaction and transactions must be deterministic

But before we focus on the design choices of Hyperledger, let us understand this notion of order execute paradigm. So most smart contract based blockchains, they actually follow the idea of what is called a state machine replication. So state machine replication, we will go more into details of that, in this session or in the next session, is the idea that if I want a system to be fault tolerant, then I have to replicate the service.

So let us say I am providing a service, let us say a database service, but I want to make sure that if my service crashes for example, then I should be able to still able to serve the customer or serve the users. So what people have done and this is more than 40 years old theory from distributed computing literature is that you make replica of the see you can think of a service as a state machine, right.

So you start with an initial state, the user interacts with you, you change the state, and you then interacts with some more, and then you change to another state and so on and so forth. So you can think of a program or a service as a state machine. So if you want to serve customer's irrespective of possibility of, let us say crash, then what you would like to do is to have a replica of the same service.

But that replica must reflect all the state changes of the original machine, because otherwise suppose I crash at some point and then the user request is routed to the replica, if the replica is still in the initial state, then replica will not be able to serve correctly the customer and the customer it will be the user and the user will be very will understand that I am not any more talking to the service I was talking to.

So in order to make it transparent to the user as well as give availability to the service, you have to have the replica always be in lockstep with the original services state transitions. So therefore, you have to replicate what is called a state machine. And this is the idea of state machine replication. Now as you will see that other than crash, there are other possibilities. For example, a particular service may be down for maintenance.

So in that case, it is not a really a crash, it is actually a known declared you know downtime for one of the component of the service. Another possibility is that the either the original or the backup replica will actually start behaving weirdly because it has been taken over by a rogue entity or it has been affected by malware, or somehow its program starts behaving strange. And in such case, we say that that is a Byzantine fault.

So there are all these different kinds of fault models people thought about when they started talking about this replicating services. And we will see that, that is where the original problem of consensus came into existence. Then you have to be in a consensus so that you are all agreeing on the next state for example. Now in case there is a fault, then also you have to keep maintaining this state transition as you interact with the user.

So this is called the active replication. So a protocol for consensus, or atomic broadcast needs to order the transactions that propagate to all the peers. So transactions may come to your system. And you have, let us say replicas, 4 replicas, or two replicas, whatever for your service. Now depending on where in the network this replicas are the transactions may come to the primary replica, or it may come to one of the secondary replicas.

Now as we saw, as we saw before, that if I give money from A to B, and then B tries to give money to C, then if B's original balance was zero, and the B to C transaction is only possible when A gave some money to B, then that ordering is very important. If the ordering is done differently, then in one case, both the transactions will succeed. In the other case, if B to C goes first, then it will not succeed, but A to B will succeed.

So as a result, if different replicas, do the transactions in different orders, then you will see the replicas are now out of state. Their states will differ. In order to avoid that, what you have to do is you have to ensure that the transactions are ordered and everybody agrees on that ordering. And if that ordering is agreed upon, then the state will also go in lockstep.

So in Ethereum, for example, what happens is, the consensus step decides that ordering, right. Remember that all of the miners are trying to create a block and in that block they choose the order of transactions and then everybody else, once they once the block has been actually created through proof of work or whatever the processes, then it is being sent to every other node and every node will eventually agree to add that as their blockchain next block and therefore, the state will always be the same.

But, one of the major issue is that ordering will never be different in different nodes, because the ordering is done by the winning the mining node. So consensus is done first in bitcoin in Ethereum and then once the consensus is done, then every other node falls in there in line with the same ordering.

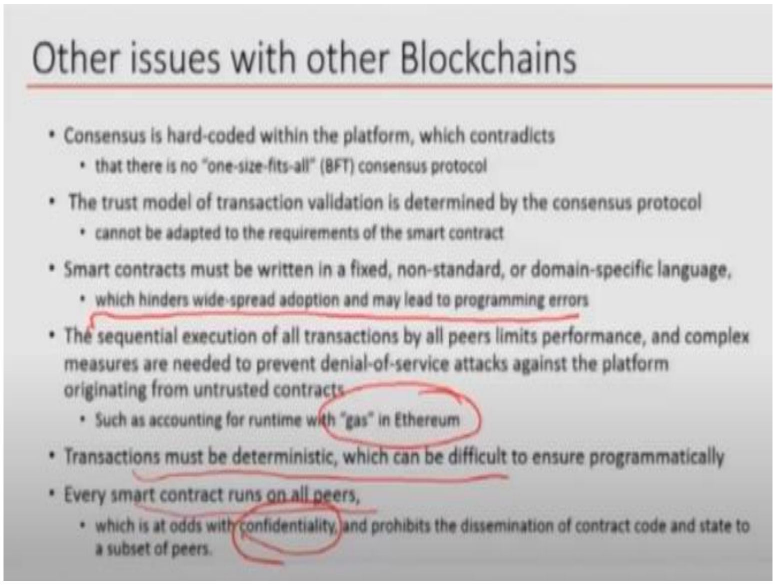
So therefore, we call it an order execute paradigm, which means that ordering is first and then execution. Execution means that every node will change their state in the same way, right. And then when the block goes to the other nodes, they will then do the transactions. They also have to remember they have to also run the same transactions and change the states and so on so forth.

So they have to do this sequentially as they get the block that wins. So this order execute architecture has been the first two generations of blockchain has been characterized by this order execute paradigm. And it requires all peers to execute the transactions and all transactions have to be deterministic. Remember that in Ethereum, we cannot have a random, call to a random number generator for example.

Because we do not want non-determinism because once the block has been decided, order has been decided, then that block goes to somebody else, they will also execute all the transactions and if there is non-deterministic execution, then the state in one node will be different from the state in the other node. So determinism is a must.

So order execute architecture is in Ethereum or in some other permission based ones like tender meat chain and Quorum. And then transactions must be deterministic. So these are the hallmarks of the earlier generation of blockchains.

(Refer Slide Time: 10:31)



Other issues with other Blockchains

- Consensus is hard-coded within the platform, which contradicts
  - that there is no "one-size-fits-all" (BFT) consensus protocol
- The trust model of transaction validation is determined by the consensus protocol
  - cannot be adapted to the requirements of the smart contract
- Smart contracts must be written in a fixed, non-standard, or domain-specific language,
  - which hinders wide-spread adoption and may lead to programming errors
- The sequential execution of all transactions by all peers limits performance, and complex measures are needed to prevent denial-of-service attacks against the platform originating from untrusted contracts
  - Such as accounting for runtime with "gas" in Ethereum
- Transactions must be deterministic, which can be difficult to ensure programmatically
- Every smart contract runs on all peers,
  - which is at odds with confidentiality, and prohibits the dissemination of contract code and state to a subset of peers.

The second issue is that remember, in bitcoin you cannot overnight change the consensus mechanism. Then you have to do some kind of a hard fork, right. Similarly, in Ethereum you cannot change the consensus mechanism, you have to do a hard fork. So therefore, the transaction we say that the consensus mechanism in this block chains are hard coded. You cannot change them and without creating forks.

In Hyperledger and more recent blockchain they have somehow extracted out the consensus mechanism so that you can actually depending on your need, you can change the consensus mechanism. So the trust model of transaction validation is determined by the consensus protocol. So trust model means what like so in case of bitcoin you do not trust any of the other nodes.

In case of Ethereum also you assume that nobody is trusted. And even these smart contracts are coming from anybody and everybody. They can actually deploy a smart contract. So the smart contract itself may not be actually, you know honest. And therefore, the trust model is that do not trust anybody. But in a permissioned blockchain since you have strong identity of each node or each user, therefore, you can decide what will your trust model be.

For example, as I was saying that if your blockchain has multiple organizations participating, then all the SBI let us say SBI, UBI they are together doing maintaining a blockchain and doing all their transactions on the blockchain. Then you can assume that the SBI employees they know each other because their identities are clear. And UBI employees know each other, their identities are clear.

So they can trust each other, trust within their group but may not trust across the group. So this is one type of trust model. Another type of trust model could be that you do not trust anybody. So you do not even trust your SBI colleagues, you do not trust your UBI colleagues. And then your mechanism for transaction validity etc. will be lot more complex. Because then now you cannot trust anybody.

But if you can trust somebody, then your mechanism for transaction whether the transaction is valid or not, etc. will be different. So now this trust model is different from the consensus trust model. The consensus trust model is about what is the fault the what is the fault model you are assuming about the consensus mechanism. So the remember the consensus mechanism is about ordering the transactions.

And so if you assume that the nodes that are participating in the consensus mechanism, some of them can be rogue or some of them could be affected by let us say malware, then you have to have a very strong consensus mechanism. If you trust all these nodes, then you can have a very simple consensus mechanism. So therefore, what you have to do is that, you have to consider two different trust models.

One model is about those who are doing transaction among themselves, how much they trust each other. And another trust model is that those nodes which are autonomous and which are doing the consensus, how much you can trust those. And in bitcoin and Ethereum these two trust models are intermingled. There is no difference in these two trust models.

But in a permission scenario, this can be very different and therefore, we have to somehow extricate the one from the other and have two different trust models and you can play with the trust model. You may have, if you have a very nice you know secure set of nodes which are doing consensus, then you can fully trust that. Whereas, since the transactions are coming from outside from clients and so on, you may not trust.

Or you may see that the all the people who are doing transaction are within your organization, you trust them. You can have a good trust, but you may say my consensus mechanism is maybe faulty. Some nodes may actually crash and so on. So I cannot trust that fully. So I have to have the right complex consensus mechanism. So you can have all kinds of mixes of the two trust models.

So other thing is that in bitcoin for example, you have to write any kind of complex logic, business logic like micro payments or escrow etc. in a nonstandard fixed language called bitcoin script language. In the case of Ethereum, you have to write them in Solidity. Now there are some few other languages which can also converted to the EVM bytecode.

But, in case of Hyperledger as I said before, you can write the smart contracts in any programming language, because there is no EVM concept or VM concept associated with them. You are basically going to execute the smart contracts inside dockers and in a regular programming environment. And this fixation with specific language is, you know hindering the wide adoption of this blockchain technology.

Because there are not many people knew Solidity, not many people who knew Go, but lots and lots of people know Java, C++, Python, etc. So you can do that. Now another issue is that in bitcoin and Ethereum etc., all once you get a block, you have to execute all the transactions in that block, even though you did not mind that block.

And so therefore, if a transaction, if a smart contract that you are executing is very time consuming, then you might get into an infinite loop or very long loop. And then that particular node will become engaged for a long time, and that can lead to what is called a denial of service attack. That node will not be able to add to the blockchain and therefore, nodes will be busy processing the last block’s transactions.

So in Ethereum this is solved by using gas mechanism, right. But in the gas, mechanism works because Ethereum has a native cryptocurrency. So you can talk about we that is the in a smallest unit of Ethereum and you can say gas costs so many weights and so on so forth and you can do that. But if you are having a permissioned blockchain in a private setting, you do not have a native cryptocurrency.

Therefore, you cannot really have this notion of gas or or incentive mechanism like in bitcoin and therefore, you cannot stop this possibilities of infinite loop and so on. And so therefore, this you cannot afford to run the smart contracts at every node. Smart contracts are only run on specific nodes, which you trust. And the smart contracts are written by specific entities that you trust.

So therefore, you assume that they will not put in an infinite loop or some kind of a very computationally intensive logic inside the smart contract. So because you are trusting so much you can afford to do that. Therefore, that is why you are writing programs in general purpose language C, C++, with which you can do such things like creating infinite loops very easily.

But you assume that smart contracts are not written by random untrusted people. It is written for specific business purpose by the entities that you know very well. Their signature is associated with the smart contract and the corresponding to that signature there is a digital identity and so on. So therefore, you can do afford to do that.

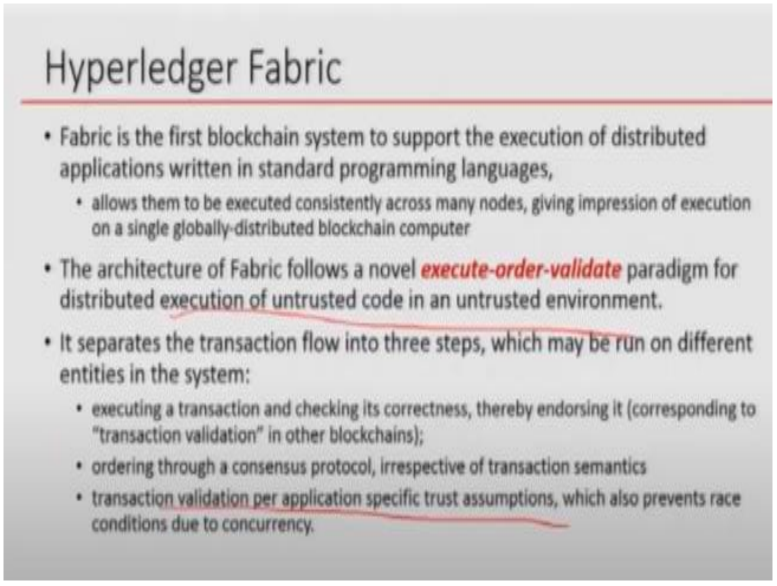
Also in those platforms like Ethereum therefore, remember that otherwise there will be divergence if the smart contracts have non-deterministic logic, because once the block comes to you being mined by one of the nodes, you have to execute the transaction. If the transaction was non-deterministic, the result you will get will be different from the other blocks what the other nodes will get, and then there will be a fork in the blockchain.

And this will be too often and therefore, you cannot afford that. So determinism is has to be inbuilt into the language in which smart contract is written. And to do that your language should be designed and that is why there is specific languages like Solidity or Ethereum you know scripting language, which do not have any construct for non-determinism.

And then, again, this we talked about earlier that every smart contract runs on every node in case of Ethereum, which means that every node needs to know your smart contract the byte code of your smart contract. And therefore, any intellectual property you have in the logic of the smart contract or any business secret as to how you process transactions, how you make decisions on the transactions etc., will be known to everybody else.

That is also not desirable. And therefore, the confidentiality of the business secrets etc., is not kept and therefore, we need to have some way of also not having a system in which smart contracts will be run at every node. So all these things that they figured out in the earlier blockchains that they want to get, make a difference from.

(Refer Slide Time: 20:58)



Hyperledger Fabric

- Fabric is the first blockchain system to support the execution of distributed applications written in standard programming languages,
  - allows them to be executed consistently across many nodes, giving impression of execution on a single globally-distributed blockchain computer
- The architecture of Fabric follows a novel *execute-order-validate* paradigm for distributed execution of untrusted code in an untrusted environment.
- It separates the transaction flow into three steps, which may be run on different entities in the system:
  - executing a transaction and checking its correctness, thereby endorsing it (corresponding to "transaction validation" in other blockchains);
  - ordering through a consensus protocol, irrespective of transaction semantics
  - transaction validation per application specific trust assumptions, which also prevents race conditions due to concurrency.

So Hyperledger has many different incarnations. So Hyperledger Fabric is the one of the most used one. So we are talking about Hyperledger Fabric. There are other implementations of Hyperledger, but Fabric is the, I would say most canonical implementation of Hyperledger, which is done by IBM. So Fabric is the first blockchain system. So to support execution of distributed applications written in any programming language.

So this gives us the ability to attract people to use it. So there is adoptability will increase because, if I have to code a business logic in C, I already know C, I do not have to learn Go or Solidity or whatever that makes the barrier to entry into this technology much lower. So the architecture of Fabric follows execute-order-validate. So earlier we were talking about order execute.

So order first execute next. Now we are talking about execute-order-validate paradigm. And we will explain what that means for distributed execution of untrusted code in an untrusted environment. So it has a transaction flow, which has three steps. First executing a transaction and checking its correctness and thereby endorsing it. So there is a step called endorsement.

So when client or the user of the application proposes a transaction, for example, give this much money from my account to somebody else's account or give this much price for the entity I am purchasing from that other entity. Then, the endorsers are certain nodes which will simulate the execution of that transaction and say If the transaction happens, then this will be the final situation.

Your account will be debited, that account will be credited or things like that. So that information along with an endorsement which is which comes in the form of a digital signature of the endorser comes back to the client or to that code that is running on behalf of the client. Then, after and this endorsement may be from one node, or it may be from multiple nodes depending on what is called an endorsement policy.

The endorsement policy is also flexible. So then, once you have got enough endorsements, as per your endorsement policy, if the endorsement policy says one endorsement is enough, that is fine. That then you are done with one endorsement if it says that it has to be a two out of these three nodes, then that also has to be gotten, things like that.

And then what you will do is that you send the send this endorsements with the possible changes in the state that is how much your balance will be, what will be deducted from your balance, what will be added to the other balance etc. to the ordering service. Ordering service will then order the transaction. Ordering service does not care about the semantics of the transaction, it does not execute the transaction.

It just do the check the read/write sets and order them. And there is some notion of a version issue also. But right now let us not worry about it. Just understand this that the transactions will be ordered and a block will be cut and that block will be sent to peers called validating peers. So then validation will be done by this validating peers.

What they will do is they will check which transaction is meaningful, which transaction is meaningless and they will mark those transactions which are meaningless. And for the rest of the transactions, it will use the read/write set to change the state replica of the entire system in its local copy and add that block to the end of the blockchain. So this is validating peers are also designated peers, not everybody in the network has to do validation.

So you see that for a transaction, not everybody executes the transaction or everybody does not need to endorse the transaction for each transaction or application. There is a transaction endorsement policy and that has to be fulfilled. Similarly, validation also is done by specific validating nodes. And when we come back, we will get into more details of Hyperledger Fabric.