On the Design and Accountability of Byzantine Fault Tolerant Protocols

Ethan Buchman
Tendermint, University of Guelph

Jan 27 2017
BPASE - Stanford
- Formal framework for reasoning about and proving properties of consensus protocols

- A general purpose abstraction over consensus protocols

- “Protocol Picker” - client provides assumptions (trust, time, failures, crypto, etc), we return optimal protocol
Contents

- Background:
  - Safety, Liveness, Asynchrony, Byzantium

- Accountability and Justification
  - Failure Detectors, Evidence

- PBFT and co.

- Application-Blockchain-Interface
Background
Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER
Yale University, New Haven, Connecticut

NANCY A. LYNCH
Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

MICHAEL S. PATERSÖN
University of Warwick, Coventry, England

Abstract. The consensus problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. In this paper, it is shown that every protocol for this problem has the possibility of nontermination, even with only one faulty process. By way of contrast, solutions are known for the synchronous case, the “Byzantine Generals” problem.
Not so fast ...
Consensus in the Presence of Partial Synchrony

CYNTHIA DWORK AND NANCY LYNCH
Massachusetts Institute of Technology, Cambridge, Massachusetts
AND
LARRY STOCKMEYER
IBM Almaden Research Center, San Jose, California
Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols
(Extended Abstract)

Michael Ben-Or

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139
Safety and Liveness
Primaries and Views

Yahya Jammeh

Adama Barrow
"Well, let's get started now we've got a quorum."
Additional phases of communication are required to detect a liar.

*Three* processes is not enough to tolerate *one* liar across a network partition.
Accountability and Justification
“Unreliable failure detectors for reliable distributed systems”

→ an abstraction over “timeouts”
→ alternative way to model synchrony


Failure Detectors

Strong Completeness: Eventually every process that crashes is permanently suspected by (the FD of) every correct process.

Eventual Weak Accuracy (EWA): Eventually some correct process is never suspected by (the FD of) any correct process.

Eventual Perpetual Uniform Trust: Eventually all correct processes always trust the same correct process.

Byzantine Failure Detectors

- Byzantine version of Chandra & Toueg
  - Omission and Commission
  - All messages contain “justification”

- PeerReview
  - Shared tamper-evident, append-only record of every node’s activity
  - Uses hash-chains and local witnesses ensure each node has one log


Byzantine Faults

Eliminated:
- Random Junk (input validation)
- Impersonation (public key crypto)

Tolerated (up to $f$):
- Omission
- Duplicity
- Fraud
Tolerance simply masks the fault

Detection exposes it and the perpetrator
Accountability
A BFT protocol is accountable if, in the event of a safety violation, it is possible for correct nodes to eventually determine exactly which nodes caused the violation
“Slasher”

https://blog.ethereum.org/2014/01/15/slasher-a-punitive-proof-of-stake-algorithm/
Conjectures:

1) All safe BFT protocols are accountable
2) BFT protocols which are accountable by detection of fraud can be transformed to ones that are accountable by detection of duplicity by including “justification” in messages
PBFT
Tendermint Performance

7 datacenters on 5 continents (AWS - c3.8xlarge)

https://github.com/tendermint/network_testing
Application Blockchain Interface (ABCI)
Somebody else’s state machine

- Zookeeper, etcd, consul
  - Fancy key-value store
  - Emphasis on distributed systems tasks (dynamic config, locking, etc.)

- Bitcoin
  - “Programmable money”
  - “Functional programming” - no state (!), contracts renewed every transaction
  - Forth like, purposefully not Turing-complete

- Ethereum
  - “Smart contracts”
  - “Contract-oriented” - stateful contracts live independently on the blockchain
  - Turing complete (Ethereum Virtual Machine)
type Application interface {
    // Info, queries, and options
    Info() ResponseInfo
    SetOption(key string, value string) (log string)
    Query(query []byte) Result

    // Validate a tx for the mempool
    CheckTx(tx []byte) Result

    // Process a block
    BeginBlock(hash []byte, header *Header, voteInfo *VoteInfo)
    DeliverTx(tx []byte) Result
    EndBlock(height uint64) ResponseEndBlock
    Commit() Result
}
type Result struct {
    Code CodeType
    Data [][]byte
    Log string // Can be non-deterministic
}

type ResponseEndBlock struct {
    Diffs [][]*Validator
}

type VoteInfo struct {
    ByzantineValidators [][][]byte
    AbsentValidators *BitArray
}
Thanks!

https://tendermint.com

https://cosmos.network

https://github.com/tendermint/tendermint