Denial-of-Service on Tendermint

Seminars in Advanced Topics in Computer Science Engineering
2016/2017

Marco Favorito 1609890

February 20, 2018

Contents

Introduction

In this report I describe an experimental Denial-of-Service attack against the Tendermint protocol, using Ethermint as ABCI application.

In Section 1 I summarize the main features of Tendermint and how it works;
In Section 2 I analyze the Tendermint protocol wrt CAP properties;

In Section 3 I describe the type of DoS attack I designed and how the byzantine node for the DoS attack has been implemented;

In Section 4 I show how the byzantine node affects the Tendermint network performances.

1 Tendermint overview

In this section we explore Tendermint in its components and its consensus algorithm.

1.1 What is Tendermint

Tendermint [1, 2, 3] is a software for Byzantine fault-tolerant (BFT) state machines replication, powered by blockchain-based consensus. It is secure, since allow to 1/3 of nodes to fail, and consistent, since every correct node agree on the same state of the application.

The two main component of Tendermint are:

- Tendermint Core: consensus engine
- Application BlockChain Interface (ABCI): enables the transactions to be processed in any programming language

1.2 How ABCI works: message types

Tendermint Core interacts with the application via a socket protocol that satisfies ABCI. The message types exchanged by nodes are many. The more important are:

- **DeliverTx**: with which every transaction is delivered in the blockchain. Application needs to validate each transaction received with the DeliverTx message against the current state;
- **CheckTx**: used for validate transactions, before entering into the mempool;
- **Commit**: used to compute a cryptographic commitment to the current application state, to be placed into the next block header.

Tendermint Core creates three ABCI connections to the application:

- for validating transactions to put into the mempool;
- for run block proposals for the consensus engine;
- for querying the app state.

1.3 How Tendermint Core works: the consensus algorithm

Tendermint Core manages the Proof-of-Stake consensus algorithm to commit the incoming transactions in the blockchain.

In this section I state the working assumptions and the consensus phases of the algorithm.

1.3.1 Working assumptions

The working assumptions, in order to allow the algorithm to work, are [1, Section 6.1: On Byzantine Consensus]:

1. **Assumption 1**: The network is partially synchronous;
2. **Assumption 2**: All non-byzantine nodes have access to an internal clock that can stay sufficiently accurate for a short duration of time until consensus on the next block is achieved; The clocks do not need to agree on a global time and may drift at some bounded rate relative to global time.
3. **Assumption 3**: At least 2/3 of the voting power is honest.

Motivations of the working assumptions

1. When the Assumption 1 fails, no consensus is possible. This result is known as the FLP impossibility result [5].
2. The Assumption 2 is needed in order to ensure that eventually the consensus procedure at a certain height is completed. Key statements to see this are:
   - Each round is longer than the previous round by a small fixed increment of time. This allows the network to eventually achieve consensus in a partially synchronous network [1, Section 6.2];
   - The asynchronous and local nature of CommitTime allows the network to maintain consensus despite drifting clocks, as long as the clocks remain accurate enough during the consensus process of a given height [1, Section 6.2];

   In other words, there is no need of global time synchronization, but the drift has to be bounded to allow the timeout mechanism to work. Eventually, the timeouts become big enough to allow the messages to be delivered in time for an enough number of nodes. Clocks do not need to be synced across validators, as they are reset each time a validator observes votes from two-thirds or more others.
3. Assumption 3 ensures *safety* and *liveness*. We will deepen the analysis in Section 2.

1.3.2 Consensus phases

There are 3 phases (**Propose**, **Prevote**, **Precommit**) plus 2 special phases, **Commit** and **NewHeight**.
A **Round** is defined as:

```
Propose -> Prevote -> Precommit
```

In the optimal scenario, the order of steps is:

```
NewHeight -> Propose -> Prevote -> Precommit+ -> Commit -> NewHeight ->...
```

Why things might go wrong? Some examples:

- The designated proposer was not online.
- The block proposed by the designated proposer was not valid.
- The block proposed by the designated proposer did not propagate in time.
- The block proposed was valid, but +2/3 of prevotes for the proposed block were not received in time for enough validator nodes by the time they reached the Precommit step. Even though +2/3 of prevotes are necessary to progress to the next step, at least one validator may have voted ¡nil¿ or maliciously voted for something else.
- The block proposed was valid, and +2/3 of prevotes were received for enough nodes, but +2/3 of precommits for the proposed block were not received for enough validator nodes.

The common exit conditions for the algorithm are:

- After +2/3 precommits for a particular block. `->` **goto** Commit(H)
- After any +2/3 prevotes received at (H,R+x). `->` **goto** Prevote(H,R+x)
- After any +2/3 precommits received at (H,R+x). `->` **goto** Precommit(H,R+x)

Now I briefly describe every phase of the state machine behind the Tendermint protocol. The specification is mainly taken from [3].

Propose (height H, round R) Upon entering Propose, the designated proposer proposes a block at height H and round R.

The proposer is chosen by a deterministic and non-choking round robin selection algorithm that selects proposers in proportion to their voting power. (see implementation). A **Proposal** is constituted by:

- a block
- an optional latest PoLC-Round < R (proof-of-lock-change) which is included iff the proposer knows of one. This hints the network to allow nodes to unlock (when safe) to ensure the liveness property

A proposal is signed and published by the designated proposer at each round.
Exit conditions:

- After timeoutProposeR after entering Propose. -> **goto** Prevote(H,R)
- After timeoutProposeR after entering Propose. -> **goto** Prevote(H,R)
- After receiving proposal block and all prevotes at PoLC-Round. -> **goto** Prevote(H,R)
- After common exit conditions.

Prevote Step (height H, round R) Upon entering Prevote, each validator broadcasts its prevote vote.

First, if the validator is locked on a block since LastLockRound but now has a PoLC for something else at round PoLC-Round where $LastLockRound < PoLCRound < R$, then it unlocks.
If the validator is still locked on a block, it prevotes that.
Else, if the proposed block from Propose(H,R) is good, it prevotes that.
Else, if the proposal is invalid or wasnt received on time, it prevotes *nil*.

The Prevote step ends:

- After +2/3 prevotes for a particular block or . -> **goto** Precommit(H,R)
- After timeoutPrevote after receiving any +2/3 prevotes. -> **goto** Precommit(H,R)
- After common exit conditions

Precommit Step (height H, round R) Upon entering Precommit, each validator broadcasts its precommit vote.
If the validator has a PoLC at (H,R) for a particular block B, it (re)locks (or changes lock to) and precommits B and sets LastLockRound = R.
Else, if the validator has a PoLC at (H,R) for *nil*, it unlocks and precommits *nil*.
Else, it keeps the lock unchanged and precommits *nil*.
A precommit for *nil* means "I didnt see a PoLC for this round, but I did get +2/3 prevotes and waited a bit".

The Precommit step ends:

- After +2/3 precommits for *nil*. -> **goto** Propose(H,R+1)
- After timeoutPrecommit after receiving any +2/3 precommits. -> **goto** Propose(H,R+1)
- After common exit conditions

Commit Step (height H) Set $CommitTime = now()$ and wait until block is received. `->` **goto** NewHeight(H+1)

NewHeight Step (height H) Move *Precommits* to *LastCommit* and increment *height*, set $StartTime = CommitTime + timeoutCommit$ and wait until $StartTime$ to receive straggler commits. `->` **goto** Propose(H,0)

In Figure 1 is depicted the message passing schema:

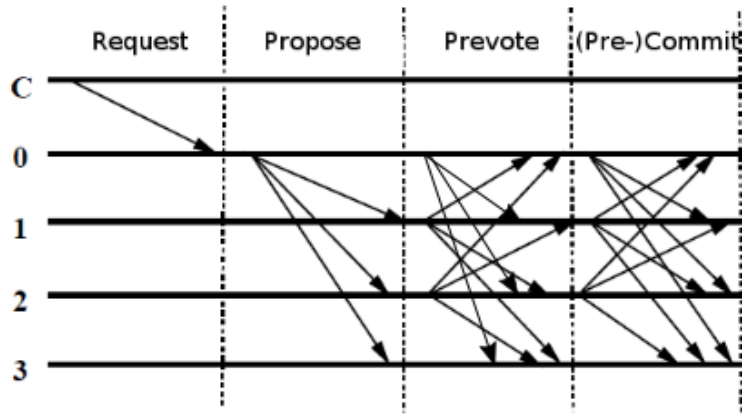

Figure 1: Message passing schema for the Tendermint consensus algorithm.

2 Tendermint and the CAP Theorem

In this section we follow the same approach used in [6].

The CAP properties (namely, *Consistency*, *Availability* and *Partition Tolerance*) in blockchain applications are interpreted as the following:

- Consistency: A blockchain achieves consistency when forks are avoided.
- Availability:A blockchain is available if transactions submitted by clients are served and eventually committed, i.e. permanently added to the chain.
- Partition Tolerance: When a network partition occurs, Tendermint validators are divided into disjoint groups in such a way that nodes in different groups cannot communicate each other.

Since a blockchain must tolerate partitions, hence CA option is not considered, we analyse the algorithms with respect to CP and AP options.

2.1 Analyzing Consistency

As stated in [1]:

> A block is considered committed when a 2/3 majority of validators sign commit votes for that block. A fork occurs when two blocks at the same height are each signed by a 2/3 majority of validators. By simple arithmetic, a fork can only happen when at least a 1/3 majority of validators signs duplicitously.

Hence, as long as there exists a majority of honest validators, in terms of their voting power, no forks can happen (see Assumption 3 in Section 1)

More formally, Tendermint ensures Safety and Liveness [1, Section 6.3 and 6.4] as long as $N \geq 3f + 1$[1]. The property are stated as the following:

- **Safety property**: If there are less than 1/3 in Byzantine voting power and at least one good validator decides on a block B, then no good validator will decide on any block other than B.
- **Liveness Property**: If there are less than 1/3 in Byzantine voting power then this protocol does not deadlock.

This property underlines the similarities with PBFT, more precisely on the optimal resiliency (see Section 3 "Service Properties" in [4]).

The differences with PBFT are [2, Section 10.2.4]):

- No fixed primary node: the proposer changes every blocks;
- The use of blocks allows Tendermint to include the set of pre-commit messages from one block in the next block, removing the need for an explicit commit message.
- Accountability guarantees when forks or some bad behaviors happen [2, Section 3.5].

2.1.1 What is Ethermint

Ethermint[2] it is an app that:

- It implements the logic of Ethereum
- It is ABCI-compliant

All consensus stuff is managed by Tendermint Core.

[1] NOTICE: numbers in the inequality must be intended in terms of voting power and not in terms of number of nodes.

[2] `https://github.com/tendermint/ethermint`

2.2 Analyzing Availability

As stated in [2, Section 3.2: Consensus]:

> After the proposal, rounds proceed in a fully asynchronous manner - a validator makes progress only after hearing from at least two-thirds of the other validators. This relieves any sort of dependence on synchronized clocks or bounded network delays, but implies that the network will halt if one-third or more of the validators become unresponsive.

In other words, if Assumption 1 does not hold (i.e. delays are unbounded), the algorithm simply halts.

2.3 Conclusions

When the assumption on the network fails, consistency is preserved while availability is given up. Hence, Tendermint can be classified as CP system, according to the CAP theorem.

3 DoS: Tendermint Evil

In this section we describe the types of Denial-of-Service attacks developed for the project, showing how I've modified the source code of Tendermint[3] to accomplish them.

For each type of attack there is a modified version of Tendermint named *Tendermint Evil - <version-name>*. In order to implement them I forked[4] the original Tendermint repository.

3.1 Tendermint Evil 'Silent'

This attack[5] is very straightforward: it simply make the byzantine node to do not send any message in the network to the other peers.

The modified source code file is `p2p/connection.go`, in which the core functions `Send` and `TrySend` have been 'neutralized': I've commented out the lines of code which actually send bytes to the other peers.

More specifically, the main changes is:

- In `Send()`:

```
-    success := channel.sendBytes(wire.BinaryBytes(msg))
+    //success := channel.sendBytes(wire.BinaryBytes(msg))
+    success:=true
```

[3] `https://github.com/tendermint/tendermint`
[4] `https://github.com/MarcoFavorito/tendermint`
[5] `https://github.com/MarcoFavorito/tendermint/releases/tag/v0.12.1.1`

- In `TrySend()`

```
-    ok = channel.trySendBytes(wire.BinaryBytes(msg))
+    //ok = channel.trySendBytes(wire.BinaryBytes(msg))
```

In simple words, we deceive the program by simulating a successful send, so the state machine execution is not affected.

This version has not been very useful for the DoS. The problem that makes the attack not effective is that the byzantine does not respond to heartbeat messages. Hence, the connections between the other peers and the byzantine node are quickly dropped because the correct nodes wisely ignore another node if it becomes unresponsive.

3.2 Tendermint Evil 'Shy'

This version[6] is a bit smarter than the previous one: it sends heartbeat messages but do not sends any block or proposals (when it is designed as proposer for the current round) and votes.

The main changes are in `consensus/state.go`:

- In `defaultDecideProposal()`[7]:

```
-    cs.sendInternalMessage(msgInfo{&ProposalMessage{proposal}, ""})
-    for i := 0; i < blockParts.Total(); i++ {
-     part := blockParts.GetPart(i)
-     cs.sendInternalMessage(...)
-    }
-    cs.Logger.Info("Signed proposal" ...)
-    cs.Logger.Debug(cmn.Fmt("Signed proposal block: %v", block))
+    //cs.sendInternalMessage(msgInfo{&ProposalMessage{proposal}, ""})
+    //for i := 0; i < blockParts.Total(); i++ {
+    // part := blockParts.GetPart(i)
+    // cs.sendInternalMessage(...)
+    //}
+    //cs.Logger.Info("Signed proposal" ...)
+    //cs.Logger.Debug(cmn.Fmt("Signed proposal block: %v", block))
+
+    cs.Logger.Info("EVIL! Do not send proposal.")
```

- In `signAddVote()`

```
-    cs.sendInternalMessage(msgInfo{&VoteMessage{vote}, ""})
-    cs.Logger.Info("Signed and pushed vote" ...)
+    //cs.sendInternalMessage(msgInfo{&VoteMessage{vote}, ""})
+    //cs.Logger.Info("Signed and pushed vote" ...)
+    cs.Logger.Info("EVIL! Do not send vote")
```

[6] `https://github.com/MarcoFavorito/tendermint/releases/tag/v0.12.1.2`

[7] Some parts have been omitted in order to help readability. Please refer to

The main purposes of this version are:

- Weaken the network: another node failure blocks the consensus algorithm (no enough voting power to commit blocks);
- Delay the commit phase: when at some height the byzantine node becomes the proposer, the algorithm is delayed about the $timeoutProposeR$ since no proposal is made. After the timeout, the round-robin algorithm select another node, and the algorithm moves to a new round (but same height).

4 Benchmarking

In this section I show the results of the experiments. The load test has been performed by Tsung[8]. The entry point for the Tendermint network setup is `ethermint-dos.py`[9], that you will find at the root of the repository for this project[10].

4.1 Setup

The benchmarking consists of the following steps:

1. Set up the network where every node is correct;
2. Start a Tsung load test session of 1 minute and 15 user per second, where sequence of requests ("Transaction" in Tsung jargon) is:
   (a) send a transaction;
   (b) do polling until the transaction is validated.

   The requests are directed to the Ethermint app, to only one node.
3. Repeat steps 1 and 2, but set up a network with one byzantine network (using Tendermint Evil 'Shy', see Section 3)

4.2 Results

In Table 1 is shown the output of one of the session tests, concerning the statistics of the requested transactions. you can notice that:

- the **mean rate** of transaction is slightly higher in the normal case than in the evil case;
- the **mean** time of a transaction is slightly higher in the evil case than in the normal case;

Name	highest 10sec mean	lowest 10sec mean	Highest Rate	Mean Rate	Mean	Count
normal_tx	6.42 sec	4.69 sec	18.9 / sec	14.42 / sec	5.23 sec	853
evil_tx	6.87 sec	4.65 sec	20.5 / sec	13.33 / sec	5.46 sec	867

Table 1: Transactions statistics.

In the graph shown in Figure 2 we can see the number of simultaneous users connected to the server. The higher the number, the lower the responsiveness of the system to validate transactions. The reader can notice that the line of the evil setup is on average higher than the one of the normal case.

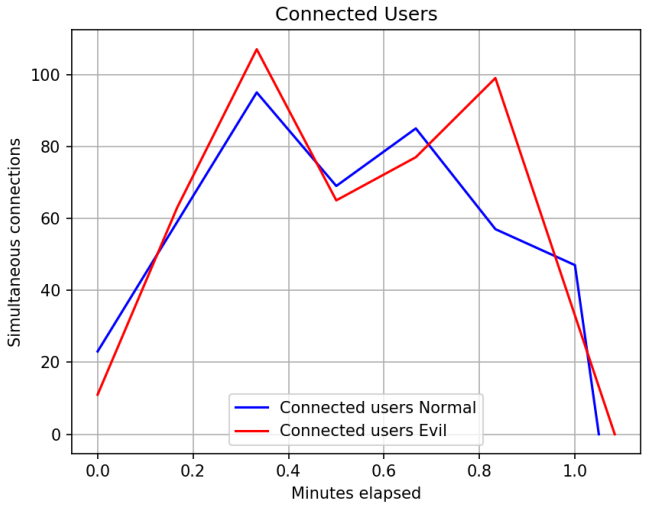

Figure 2: Simultaneous connected users

5 Conclusions

In this project I tried to perform a simple DoS attack against the Tendermint protocol. In Section 1 I briefly describe the main Tendermint features, while in Section 2 I analyzed the Tendermint algorithm froma theoretical point of view, using the CAP Theorem. In the experiment, the byzantine node does not send proposals and votes. As explained in Section 3.2, the purpose is to delay the consensus algorithm. In Section 4 I showed quantitatively how this type of DoS attack affects the Tendermint/Ethermint network, leveraging Tsung capabilities.

[8] `http://tsung.erlang-projects.org/`
[9] `https://github.com/MarcoFavorito/ethermint-dos/blob/master/ethermint-dos.py`
[10] `https://github.com/MarcoFavorito/ethermint-dos`

References

[1] Tendermint: Consensus without Mining

[2] Tendermint: Byzantine Fault Tolerance in the Age of Blockchains

[3] Tendermint Read the Docs

[4] Practical Byzantine Fault Tolerance

[5] Impossibility of Distributed Consensus with One Faulty Process

[6] PBFT vs Proof-of-Authority: Applying the CAP Theorem to permissioned Blockchain