CSC2125: Homework #5

Due on October 22

Fan Long

Thomas Hollis

Problem 1

Suppose we run PBFT algorithm on a cluster of 21 machines. What is the maximum number of machines we will be able to tolerate for failure simultaneously?

Solution

The number of simultaneous machines with Byzantine faults that can be tolerated by PBFT is given by the following equation provided in the whitepaper: $[\frac{n-1}{3}]$. Hence here we would have: $[\frac{n-1}{3}] = \frac{20}{3} = 6.\dot{6}$. Therefore, the maximum number of machines that could exhibit Byzantine behaviour is 6, because for 7 machines or higher, PBFT would not be able to tolerate this without failure.

Problem 2

A node in PBFT waits for prepare/commit messages from two thirds of nodes (replicas) during the prepare/commit stages. What would happen if the node instead waits for prepare and commit messages from only a simple majority of other nodes? Would the modified PBFT be secure? If not, please explain with a counter example.

Solution

To answer this question, we need to provide some background explanation of the PBFT solution first, in particular on the prepare and commit stages. The network has a size of $3f + 1$ where f is the maximum number of faulty nodes (as seen in the previous exercise). Indeed, figure 1 shows that when a client makes a request the primary sends a "pre-prepare" message to all the $3f$ backups. The backups then in turn send a "prepare" to each other (and to the primary). This is to make sure that they are all ready to commit to the same order of operations within a single view v. Once enough prepare messages are diffused the network agrees on the order and gets ready to make commits. Then the answer of these operations must be agreed upon within the network (even if a view change occurs). Thus, each node sends commits to all other nodes in the network to ensure they all agree before committing locally in the commit stage. At this point all honest nodes have agreed both on the order of operations and on the results (independently of view changes). Only then can the $3f + 1$ nodes go ahead and send their reply to the client. Once the client receives the replies he needs to wait until he has received at least $f + 1$ replies that all agree to make sure that he has the correct reply. This is because even if there are f nodes maliciously reporting the wrong result in the group of $3f + 1$, there will always be at least one honest node to contradict them nullifying their attack.

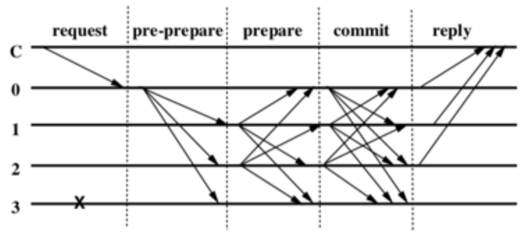

Figure 1: Normal Case Operation

In this case, at the prepare/commit stages, we must prove $f + 1$ non-faulty nodes have agreed to commit the same information independently of view changes. Indeed, we want to have a majority consensus among the honest nodes not among all nodes (including byzantine ones). Thus, to be sure we have $f + 1$ non-faulty nodes we need to have $2f + 1$ nodes (two thirds of nodes) to account for a maximum of f faulty nodes. Thus, we need to wait $2f + 1$ in the prepare stage to agree on the order and $2f + 1$ in the commit stage to agree on the final result and state.

If we only rely on $\frac{3f+1}{2}$ nodes (i.e. half of the nodes) to agree to commit the same information, we could end up with different groups committing different information breaking the PBFT system. This in turn could lead to incorrect results (and inconsistent states in the backups) and thus the PBFT system would therefore no longer be secure.

A simple counter example to demonstrate this would be as follows: an attacker controls f nodes. At the prepare stage you wait for $\frac{3f+1}{2}$ nodes (half) rather than $2f + 1$ nodes (two thirds). This means the network could move on to the next stage but with different groups considering different orders. One group would want to commit to order 1 and another would want to commit in order 2. This lack of consensus would ripple through to the next stage, the commit stage. Indeed, this would eventually cause the states to be broken (inconsistent) resulting in inconsistent replies to the client.

Problem 3

PBFT relies on the primary node to send out pre-prepare messages to drive the consensus process. What would happen if the primary node is malicious or fails? How does PBFT handle this situation?

Solution

PBFT has a number of protections against the primary node being malicious or failing.

The first approach that a malicious primary could do is exhaust the space of sequence numbers by selecting a very large one. However, this is protected in PBFT by using watermarks h and H.

Another approach for the malicious primary would be to simply stop multicasting requests to the group. However, if this occurs, the primary will eventually be suspected to be faulty by enough replicas to cause a view change.

If the primary node fails completely, it will be look like a malicious node which is purposefully not acting. Therefore, like in the previous situation, the replicas will suspect a Byzantine faulty primary and eventually trigger a view change.

This view change allows the system to continue to progress even when the primary fails or is malicious. Indeed, this corresponds to a fundamentally desirable property of concurrent systems known as liveness (this as well as the safety property helps prevent deadlocks).

View changes are triggered by timeouts which prevents indefinite waits. What happens is the backup (which checks on the primary) starts a timer when it receives a request (and when the timer is not already running). The backup then stops the timer when it is no longer waiting to execute the request. This process will repeat for new incoming requests. However, if the timer of a particular backup expires in the current view, the backup begins a view change to move the entire system forward by one view (i.e. move to view $v+1$). At this point it stops accepting messages (except checkpoint, view change and new view messages) and multicasts the following message to all replicas: $<$VIEW-CHANGE$, v+1, n, C, P, i >_{(\sigma_i)}$

When the primary receives $2f$ valid view-change messages for $v+1$ it will multicast the following message to all replicas (before making various changes to his log): $<$NEW-VIEW$, v+1, V, O >_{(\sigma_p)}$

However, if the primary does not cooperate because it is malicious or fails, a supermajority of honest nodes can decide that the primary is faulty and choose the new primary (based on who is next in line in the round robin primary selection process) by themselves triggering a new view.

Problem 4

Could PBFT skip the commit messages? Suppose nodes in PBFT commit-local directly after receiving $2f+1$ prepare messages. What could go wrong? Please explain with a counter example (Consider the case where a view change happens immediately after one or two nodes commit-locally for a request).

Solution

We know that $3f+1$ is the minimum number of replicas that allows asynchronous systems to provide safety and liveness properties (for f faulty replicas), as proved by Bracha and Toueg. However, additional replicas do not improve resiliency but do degrade performance. Indeed, in PBFT the main protocol used to ensure liveness is the protocol of view changes.

Here we are examining the impact of skipping the commit stage. This means once each node agrees on the order of operations (i.e. receive $2f+1$ prepare messages) they commit-local directly without checking their commits with other nodes. This is an issue because the pre-prepare and prepare stages are used to ensure that requests are totally ordered within a view v while the commit stage ensures requests are ordered even after a view change.

Therefore, if a PBFT system was to skip these commit messages, the main thing that could go wrong is that this may cause the system to violate consistency. This is because some nodes would commit to some result before changing view while other nodes would not so there is no guarantee of consistency in the new view.

This situation can be construed when view changes occur. Let us propose a case where a view change happens immediately after one or two nodes commit-locally for a request. At this point all nodes have agreed on the total order of requests within view v. However, if a view change occurs some nodes may commit-locally before the view change while others may not. In normal cases if the commit messages are sent, the network can recover and agree to consistent local commits even after a view change. This is because there will always be at least one node that will bridge the gap. However, if instead nodes all commit-locally skipping the commit stage, they would result in inconsistent states. This would break the systems overall consistency.

Problem 5

A permissioned blockchain is a blockchain system in which the membership of the blockchain is fixed and known to all participants. The membership may be managed offline by a potentially centralized organization. Explain how PBFT could be used to implement such a permissioned blockchain.

Solution

Lets take the example of a hypothetical permissioned blockchain B. Lets supposed that B is primarily used to manage the supply chain of company C. This is quite popular at the moment as businesses often want to know who is behind each node so they can monitor activity tied to a particular identity. Transactions on blockchain B could involve a variety of entities including logistics partners, financing banks and other vendors.

If one of the members of blockchain B gets hacked, he will begin to act as a faulty node. The hacker may try to bring down the company C by writing irreversible data onto their blockchain B. This property of non-repudiation in blockchains, while usually highly desirable, can sometimes be an issue. As a business, this irreversible behaviour is highly undesirable if it is undertaken by a rogue node or rogue group of byzantine nodes. Thus, to prevent irreversible damage, PBFT could be used to temporarily nullify the actions of a malicious node until the system can recover. Indeed, in permissioned blockchains since the identity of nodes is known, the company running blockchain B would be able to identify the byzantine node and let them know that have been hacked. Blockchain B could then survive until the hack is patched and normal activity is resumed.

Similarly, if one of the members or a group of members of blockchain B does not act according to a legally binding agreement, their abuse of contract can be temporarily blocked until legal action is taken. This can be directly compared to the Byzantine generals problem because we can clearly see how a company would benefit from using PBFT to ensure all of its members attack at the same time so to speak.

A third possibility would be to use PBFT to separate public users from full nodes with consensus power. Here we can go in a bit more technical detail. Suppose blockchain B was used to record transactions. One could implement PBFT such that each public user maps to a client in PBFT and each transaction validator maps to a replica in PBFT. Like in PBFT we would have pre-prepare, prepare and commit messages such that blockchain B would become a system which can be publicly queryable but which can be managed only by trusted and known entities. This seems to fit in perfectly with the permissioned blockchain model.

In fact, the Hyperledger Fabric project (hosted by The Linux Foundation) uses PBFT for their permissioned blockchains. Their main target is to help industry through open sourcing modular blockchain architectures which is why they received so much industry backing from IBM, Intel and Cisco.

Similarly, Tendermints white paper proposes a method of implementing PBFT into blockchains. However, it is not particularly well written. Indeed, the conclusion simply states "Tendermint is awesome. The future is now." which has raised many eyebrows among the blockchain engineering community.

Problem 6

Suppose the number of replicas in the system is N. During the normal-case operation in PBFT, roughly how many messages will be broadcast for each request? Suppose the broadcast is implemented via point to point transmission between nodes one by one. What is the total number of transmitted messages for processing each request? (Answer in Big O notation like O(1), O(N), O(N^2), ...)

Solution

From figure 1, we can estimate the total number of messages broadcast for each request in a network with N nodes (including the primary) as follows.

Step 1 (request): The client sends a message to the primary with his request - O(1) messages since 1 message is sent (assuming the client is not faulty).

Step 2 (pre-prepare): The primary sends a message to each backup with the request - O(N) messages since $N-1$ messages are sent (assuming primary is not faulty).

Step 3 (prepare): The backups all send a message to each other and the primary - O(N^2) messages since $(N-1)\times N$ messages are sent (assuming no unresponsive backup nodes).

Step 4 (commit): The backups and the primary all send another message to each other - O(N^2) messages since $N \times N$ messages are sent (assuming no unresponsive nodes).

Step 5 (reply): The backups and the primary all send one message to the reply with their answer - O(N) messages since N messages are sent (assuming no unresponsive nodes).

Since none of the messages transmitted in steps 1 to 5 contain other messages inside them (unlike in view changes), the total transmitted messages are: O(1) + O(N) + O(N^2) + O(N^2) + O(N) $\approx$ O(N^2)

Therefore, the total number of transmitted messages for processing a single client request is of the order O(N^2).

Problem 7

Suppose the number of replicas in the system is N. Suppose the broadcast is implemented via point to point transmission between nodes one by one. What is the total number of transmitted messages during a view-change in PBFT? What is the size of each message during a view-change? (Answer in Big O notations like O(1), O(N), O(N^2), ...)

Solution

From the original PBFT paper, we can estimate the total number of messages broadcast for a view change in a network with N nodes (including the primary) as follows.

Step 1. Backup node timer expires in view, backup stops accepting messages and multicasts a view-change message to all replicas (which contains itself sets of messages) - O(N) messages since $N - 1$ messages are sent (assuming backup node is not faulty).

Step 2. Primary node eventually receives $2f$ valid view-change messages and multicasts a new-view message to all other nodes (which contains itself sets of messages) - O(N) messages since $N - 1$ messages are sent (assuming the primary is not faulty).

Thus, total transmitted messages are: O(N) + O(N) = O(N)

However, since each message in step 1 and 2 contains multiple messages, the size of each of these messages are:

Step 1. Message is of size O(N) because:
- Set C: $2f + 1$ checkpoint messages thus O(N)
- Set P of sets P_m: 1 pre-prepare message + $2f$ prepare messages thus O(1)

Step 2. Message is of size O(N^2) because:
- Set V: $3f$ view change where each view change is O(N) thus O(N^2)
- Set O: variable size thus can assune O(N) worst case

Therefore, the overall size of the messages in step 1 and 2 is O(N^2).

Therefore, the total number of transmitted messages for processing a single view-change is of the order O(N) and these messages typically have a size of O(N^2). This could therefore be interpreted as having an overall overhead of O(N^3).

Problem 8

Given the above analysis, explain why it might be impractical to use PBFT to implement a large scale blockchain.

Solution

It may be impractical to use PBFT to implement large scale blockchains because of scaling issues.

As we have seen in the previous question, the space complexity of the messages is O(N^2) for a client request and O(N^3) for view changes. More specifically, as the number of nodes N grows, the number of messages sent for processing a request increases quadratically while the number of messages sent for view changes increases cubically. If we need to undertake many such requests and view changes per year this overhead is likely to clog up the network speed. This is indeed an issue for large scale blockchains (in particular for cryptocurrencies) as the write speed (or transaction speed for cryptocurrencies) is often a highly undesired bottleneck that users want to minimise.

This bottleneck has been at the root of many highly provocative criticisms and debates about protocol changes (for example the infamous BTC block size controversy). Indeed, the sheer overhead and inefficiencies of blockchains are the main reason why most data today is still stored in regular non-distributed and non-blockchain-based structures. So for this reason it might be impractical to use PBFT to implement a large scale blockchain.

Also, it is worth noting that PBFT does not protect against all types of attacks. Indeed, the main assumption is that the Byzantine nodes do not exceed $[\frac{n-1}{3}]$. If, for example, there is a software bug that renders all the nodes of a blockchain vulnerable to be taken over by an attacker and turned to Byzantine nodes, then PBFT cannot protect against this type of attack. Indeed, such bugs are unfortunately all too common because of poor programming (see ERC20 token vulnerabilities). In addition, many blockchains are today somewhat centralised due to the presence of mining pools with single users owning sometimes close to one third of the hash rate. This would make any PBFT protection somewhat unnecessarily impractical.