A Generic Byzantine Fault-Tolerant Algorithm with Commutative Commands

An implementation on a simulated network

Alexandre Messmer

January 2022

Distributed Computing Laboratory
EPFL
Switzerland

Acknowledgments

I would like to warmly thank my supervisor, Manuel Vidigueira, who made this project possible. He helped me from the programming side (which can appear very tough in Rust at first sight) to data analysis. He always has good tips, and I liked to learn from his experience. I had the chance to use one of his libraries developed at EPFL, which proved to be more than necessary. I also want to express my sincerest gratitude to Rachid Guerraoui and the DCL team, who offered me this opportunity.

Abstract

Consider a distributed computing system that manipulates data. System members, commonly denoted as peers, exchange information across a network. In particular, any peer wishes to trust others and be sure that it is dealing with the correct information. However, this idyll slowly disappears when we enter the practical deployment of such networks. In reality, distributed computer systems suffer countless errors. Components may enter a faulty state and behave arbitrarily, even fail. These peers may appear failed only for certain peers, and the information about their behavior is inconsistent inside the network. It defines one of the most common failures, the Byzantine failure. Peers thus need to know which components have failed. They, therefore, need to reach a consensus regarding the network state. How can we achieve such an agreement, and how can we use it to make our system byzantine resilient? Byzantine fault-tolerant protocols provide efficient algorithms to deal with this problem. We will analyze and simulate such a protocol, detailed in the paper *Byzantine Fault-Tolerance with Commutative Commands* written by P. Raykov, N. Schiper, and F. Pedone. It uses a generic broadcast protocol and machine state replication to deliver commutative commands efficiently.

1 Introduction

As of today, we mostly rely on a centralized banking system. As customers, we fully trust our bank to use our money as it declares. We assume that no one inside the banking corporation is malicious, and if it were to happen, the government is there to bring back the order. Federal institutions also prevent such problems with laws. However, as the reader can notice, the customer needs to trust a third party. When someone gives a company his trust, he expects that it behaves according to their contract. But what if it is not the case? And what if we can't trust the government either? A solution would be to distribute this notion of confidence. If we take a closer look, the bank is a centralized database that holds accounts with customers' balances. A solution can be to decentralize those databases to separate the bank from the user accounts. Cryptocurrencies adopt such models to remove the trusted third party. Each user has a log file (the blockchain) of the transactions that happened, and he is the only one who knows what he has. The log files allow users to agree on some transactions or not. However, those databases cannot be completely reliable, and they need to be byzantine resilient. The classical

way to proceed is to reach a consensus about who is working correctly and then process the data. But how does it work?

Let's make a small analogy[1] to describe the problem. Consider a castle that a kingdom wishes to conquer. The King sends several armies, led by generals, without coordinating them to surround the target. Even if most generals are loyal to their King, some may want to put him down and won't hesitate to confuse others. The generals need to agree on an attack, but they can only communicate through messages. We assume that messages are signed and delivered. In addition, if every general decides to attack together, they will get the castle. The traitors will only send confusing information, but they will attack accordingly to the battle plan. They thus need to agree on a coordinated attack (or a retreat) even in the presence of dishonest generals (they are the faulty components). They will eventually broadcast messages to others generals, and depending on the content of the received messages, each will get to the same plan. They will reach a consensus. In this case, the distributed army becomes byzantine resilient since generals won't agree on a different battle plan in the presence of byzantine failures.

Getting back to our distributed system, the first key to handle such failures is thus a way of processing messages to reach a consensus. We will define this abstraction as the Recovery Consensus. In the case of the Byzantine Generals problem, we only need to treat one decision. However, our system will need to handle multiple instructions, and we cannot afford computers' unavailability. A solution to increase service availability is to use state machine replication. We will implement a Byzantine Fault-Tolerant algorithm with commutative commands. The servers rely on a generic broadcast algorithm to execute commands. It uses a Recovery Consensus to order non-commutative ones. In the best case, this algorithm only needs two communication delays.

2 A BFT protocol with commutative commands

We will briefly define the system and its requirements and then describe the Recovery Consensus, the BFT generic algorithm, and its implementation

[1]This analogy is an adaptation to the commonly known Byzantine Generals problem. It was defined in 1982 in [2].

with state machine replication.

2.1 System definition

The system is composed of an unbounded number of clients that issue commands. There are n replicas that process these commands, and at most f of them are byzantine (i.e., they can behave arbitrarily). We do not consider malicious failure (i.e., an adversary controls the byzantine replicas), but this protocol works in this case as well. Peers communicate with asynchronous messages, which are signed and encrypted. We assume the existence of an atomic broadcast protocol that broadcast messages in order through the system, even in the presence of crash failures (i.e., a peer stops functioning). The network is fully connected and quasi-reliable, i.e., only messages sent by byzantine replicas are not necessarily delivered.

2.2 Recovery Consensus

Recovery consensus is an algorithm used by the generic broadcast protocol to order non-commutative messages between replicas. We define the commutativity of messages by an equivalence relation $\sim$ on a set $\mathcal{M}$. From now one we will say that two messages conflict if they do not commute. Each replica i can propose two sets: one without any conflicting element (denoted $NCSet_i$) and another with elements that conflict with a least one in the first set (denoted $CSet_i$). More formally,

$$\forall m, m' \in NCSet_i : m \not\sim m' \tag{1}$$

$$\forall m \in CSet_i, \exists m' \in NCSet_i : m \sim m' \tag{2}$$

Once it receives n_{chk} valid[1] and unique[2] proposition (n_{chk} is a parameter of the system such that $n_{chk} \leq n - f$), it decides on two sets $NCSet$ and $CSet$ as follows:

- if an element m appears in the majority of the non-conflicting sets, then it is in $NCSet$.
- otherwise, it is in $CSet$.

[1]A proposition is valid if the proposed sets $NCSet$ and $CSet$ satisfies (1) and (2) and $NCSet \cap CSet = \emptyset$

[2]A proposition is unique if we only received it once for a given signature. Note that a byzantine replica cannot modify its digital signature.

For some mathematical requirements, recovery consensus only works under the condition that $n > 5f$, where n is the number of replicas and f is the number of byzantine ones (the reader is invited to read [1], section 3, for more details). The algorithm $\mathcal{C}_{absign}$ from [1] is a pseudo-code for the Recovery Consensus:

Algorithm $\mathcal{C}_{absign}$
Process p_i Recovery Consensus algorithm with atomic broadcast and signatures

1: Procedure proposeRC($NCSet_i, CSet_i$)
2: A-Bcast($\langle NCSet_i, CSet_i \rangle_{\sigma_i}$)
3: **wait until** [$|GS| = n_{chk} : GS \stackrel{def}{=} \{(NCSet_j, CSet_j)$ | A-Delivered unique and
4: valid $\langle NCSet_j, CSet_j \rangle_{\sigma_j}$ from p_j }]
5: $NCSet \leftarrow \{m \mid \exists \lceil \frac{n_{chk}+1}{2} \rceil \ NCSet_j : (NCSet_j, \cdot) \in GS$ **and** $m \in NCSet_j\}$
6: $CSet \leftarrow (\bigcup_{(NCSet_j, CSet_j) \in GS} NCSet_j \cup CSet_j) \setminus NCSet$
7: decideRC($NCSet, CSet$)

2.3 BFT generic broadcast with machine state replication and speculative execution

The protocol is made of two actors, that both run a different protocol. On the one hand, when a client wishes to execute a command, he broadcast it to all replicas and wait for a certain number of acknowledgments. Once he receives enough acknowledgments (a parameter n_{ack} defines this number), he obtains the result of its command. On the other hand, the replica process in rounds. A round is composed of one acknowledgment phase and one check phase. In the acknowledgment phase, the replica speculatively executes non-conflicting commands (i.e., it does not wait for an agreement), adds them to its pending set, and broadcasts them to other replicas. Note that the replica must need to roll back some of them in case a conflict happens later. When it receives a command that conflicts with its pending set, it starts the check phase. The replica eventually broadcasts its pending set and proposes the commands it has already executed as the non-conflicting set for the Recovery Consensus. Once the consensus among non-conflicting sets and conflicting ones is reached, the replica eventually rolls back conflicting commands that it executed to order them later. It executes the non-conflicting ones and then executes the conflicting commands in the same order as the other replicas. The check phase ends when every command has been executed, and the protocol starts a new round. Each replica keeps track of every message received or delivered in previous rounds to avoid corruption by byzantine replicas. In

Algorithm $\mathcal{SMR}_{client}$
Client c algorithm

1: To execute command m:
2: send($\langle m \rangle_{\sigma_c}$) to all replicas
3: **wait until** [$\exists\, k$, s.t. received from different replicas
4: n_{ack} $\langle k, m, res(m), \text{ACK} \rangle_{\sigma_{r_ic}}$ **or** $f + 1$ $\langle k, m, res(m), \text{CHK} \rangle_{\sigma_{r_ic}}$]
5: **return** res(m)

Figure 1: The client protocol

the best case, i.e., when clients only issue commutative commands, the protocol allows execution in two communication delays, which is way faster than other BFT protocols (such as PBFT or HQ). The two algorithms SMR_{client} and $SMR_{replica}$ are the two pseudo-code proposed by [1]:

We propose a simplified implementation of this protocol in the next section. In particular, since this protocol relies on an existing atomic broadcast protocol, we did not have the time to fully implement the Recovery Consensus. A mock version of it was made to analyze the protocol performances. The key idea was to compare its performance to an atomic broadcast protocol, which can be simulated with only conflicting commands.

3 Implementation[1] of the protocol

To implement such a protocol, we first need to create a local network that satisfies the description given above. We used the Tokio library to manage the asynchronous part and the Talk library to create a network of peers that communicate with encrypted and signed messages.

3.1 The network

Our network is based on the UnicastSystem abstraction provided by the Talk library. The UnicastSystem creates a set of n triplets of an identity key (i.e., a unique identifier for a peer), a sender, and a receiver. A peer, in its abstraction, is made of one triplet. Since peers need to handle asynchronous

[1]The implementation is publicly available in the following repo GitHub: https://github.com/AlexandreMessmer/generic-byzantine-fault-tolerance. Note that the documentation is not exhaustive. The tests provided should be sufficient to understand how to use it.

Algorithm $\mathcal{SMR}_{replica}$
Replica r algorithm (the differences with Algorithm $\mathcal{PGB}$ are highlighted in gray)

1: Initialization:
2: $Received \leftarrow \varnothing$, $G\_del \leftarrow \varnothing$, $pending^1 \leftarrow \varnothing$, $k \leftarrow 1$, $Res \leftarrow \varnothing$
3: **when** receive($\langle m \rangle_{\sigma_c}$) **do** {Task 1a}
4: $Received \leftarrow Received \cup \{\langle m \rangle_{\sigma_c}\}$
5: **when** receive(k, $pending_j^k$, ACK) **do** {Task 1b}
6: $Received \leftarrow Received \cup pending_j^k$
7: **when** receive(k, S_j, CHK) **do** {Task 1c}
8: $Received \leftarrow Received \cup S_j$
9: **when**$\big(Received \setminus (G\_del \cup pending^k) \neq \varnothing\big)$ **do** {Task 2}
10: **if** $(\forall\, m, m' \in (Received \setminus G\_del) \;:\; m \not\sim m')$ **then**
11: **for each** $m \in (Received \setminus (G\_del \cup pending^k))$ **do**
12: $res(m) \leftarrow$ execute(m), $Res \leftarrow Res \cup (m, res(m))$
13: send($\langle k, m, res(m), \text{ACK} \rangle_{\sigma_{rc}}$) to client($m$)
14: $pending^k \leftarrow Received \setminus G\_del$
15: send(k, $pending^k$, ACK) to all replicas
16: **else**
17: send($k, (Received \setminus G\_del)$, CHK) to all replicas ▷ start of CHK phase
18: proposeRC($k, pending^k, (Received \setminus (G\_del \cup pending^k))$)
19: **wait until** decideRC($k, NCSet^k, CSet^k$)
20: **for each** $m \in pending^k \setminus NCSet^k$ **do**
21: $rollback(m)$, remove($m, res(m)$) from Res
22: **for each** $m \in NCSet^k \setminus G\_del$ **do**
23: **if** $m \notin pending^k$ **then** $res(m) \leftarrow$ execute(m)
24: send($\langle k, m, res(m), \text{CHK} \rangle_{\sigma_{rc}}$) to client($m$) ▷ $res(m)$ is retrieved from Res if needed
25: **in ID order: for each** $m \in CSet^k \setminus G\_del$ **do**
26: $res(m) \leftarrow$ execute(m), send($\langle k, m, res(m), \text{CHK} \rangle_{\sigma_{rc}}$) to client($m$)
27: $G\_del \leftarrow G\_del \cup NCSet^k \cup CSet^k$
28: $k \leftarrow k + 1$, $pending^k \leftarrow \varnothing$, $Res \leftarrow \varnothing$ ▷ end of CHK phase
29: **end if**

Figure 2: The replica protocol

messages, they loop as long as the network exists and communicate with the Talk library. In particular, the sender and receiver present a similar API from the Tokio channel. When a sender issues a message to an identity key with *spawn_send()*, the receiver with this identity key can receive it with the *receive()* method. The role of our network is then to create the peers and make them run. It can be set up with a NetworkInfo instance that holds the information. In particular, it describes the number of clients, the number of replicas, how many of them are byzantine (denoted as faulty in the library), the parameters of the protocol (i.e., n_{ack} and n_{chk}), and the transmission delay inside the network. Indeed, since the unicast system is a local system, we do not expect any transmission delay. Thus, we will need to simulate it. Since clients run on different threads and not on computers, we can't input commands directly from them. We thus need to create this behavior. The network has Tokio channels that enable him to send instructions to clients. The Instruction enumeration is composed of two elements. The first is the execution of a given command, denoted as *Execute(cmd)* (where *cmd* is the command to execute). It emulates the client's willingness to execute the command by itself. The second one is the shutdown order. Once the network has decided to stop, he broadcast this *Shutdown* to every peer to properly close them (e.g., write state to some file). The network also assigns each peer a unique identifier in increasing order from 0. We need to keep track of the identity key of each peer to send messages later on. Thus, each network member holds a table of identities (a hash table) that converts the local identifier (inside the network) and the identity key provided by the unicast system.

3.2 The peer: how do we define interactions?

Peers interact inside the network and have their own identities. They run indefinitely as computers. Their definition is separated into three abstractions:

3.2.1 The Runner trait

The first one is the Runner abstraction. A peer defines a *run()* method that needs to be called to start the peer. When a peer is running, it concurrently receives instruction from the network and messages from other peers. When it receives an execution instruction for a command, it starts a SMR_{client} protocol with this command (as if it decided it on its own). When it receives a message from other peers, it handles it with the second abstraction, the *Handler*. When it processes messages, it might need to send messages to

other peers or send feedback to the network (e.g., in case of failure). This part is managed by a third abstraction: the *Communicator*.

3.2.2 The Communicator

The communicator abstract the way the peer sends messages inside the network. It defines two methods:

- *send_message()*: this method is responsible for simulating the transmission delay and then sends the message to the targeted peer. It uses the sender provided by the *UnicastSystem* and waits for several milliseconds before sending the message. The transmission delay is distributed over a Poisson distribution with the rate given in the network information. We did it deliberately since transmission delay is never constant inside a network. We can thus expect congestion.

- *send_feedback()*: this method enables the peer to give feedback about the current execution to the network. In particular, since we do not control each peer, we cannot see the result of the command nor the failure of the algorithm at any point in time. We can then keep control of the network even if we cannot access each peer.

3.2.3 The Handler trait

The Handler trait defines the peer's behavior upon receiving messages. This defines the peer to be a client, a replica, or a faulty replica. To avoid duplication of code, the Handler is given to the peer as a trait object. The peer uses dynamic dispatch to use the correct Handler. We defined the three following handlers:

- The *ClientHandler*: it defines the behavior of a client. In particular, messages are treated as in the SMR_{client} algorithm provided in section 2. The client additionally defines a queue of pending requests, that keeps track of which commands are currently being executed.

- The *ReplicaHandler*: it defines the behavior of the replica. The implementation follows the algorithm $SMR_{replica}$ provided in section 2 and essentially manages the messages with the same data structures. In particular, we used the *BTreeSet* library to efficiently manage sets. Even if some optimizations (e.g., we could cache the sets $Received \setminus G_{del}$ and $Received \setminus (G_{del} \cup pending^k)$) were possible, we did not implement them. We assumed that these operations' costs are negligible when the transmission delay reflects reality (e.g., 100ms). According

to [1], we can set n_{ack} to be the same as n_{chk}, with the upper bound $n_{ack} \leq n - f$. By default, we will set n_{ack} to $n - f$. The replica can invoke a Recovery Consensus instance with a *propose(k, NCSet, CSet)* method, that blocks until the consensus is reached. The only difference is the implementation of the Recovery Consensus protocol.

- The *FaultyReplicaHandler*: defines the behavior of a byzantine replica. It decorates a *ReplicaHandler* so that we can use the *ReplicaHandler*'s methods with arbitrary values. By default, the byzantine failure is the crash of the replica. She stops functioning (i.e., she does not respond nor send any messages). We can configure it to behave arbitrarily (e.g., duplicate commands, modify the round number, etc.).

3.3 A mock Recovery Consensus

The Recovery Consensus protocol requires the existence of an atomic broadcast protocol. However, due to a lack of time, we did not implement it as in [1]. Instead, we propose a parametrizable (e.g., the consensus duration) mock version of it with an instance called the *Coordinator*. In essence, the difference with C_{absign} protocol is that the Coordinator decides the non-conflicting set and the conflicting set (instead of having each replica deciding on its own). We implement it as follows:

1. Each peer can propose (with the *propose()* method) sets of commands concurrently to the *Coordinator* through a multi-producer-single-consumer Tokio channel.

2. Once he receives n_{ack} valid pairs of sets, he decides on two sets $NCSet$ and $CSet$ as in the C_{absign} algorithm.

3. He then delivers the pair to every replica through a Tokio broadcast channel. We also simulate the consensus duration at this step of the algorithm.

This implementation guarantees to provide the same result as the Recovery Consensus protocol from [1].

3.4 The Banking system

The last step of our implementation is to create a small system that executes mostly commutative commands. We will stay in the financial sphere. Consider a simplified banking system where customers can open saving accounts. Each customer can have only one where he deposits and withdraws

money (he also sees how much he has saved). Because he is saving money, we assume that the customer's balance never goes below zero. We simplified the model since customers can't transfer money to another account. Instead, they need to withdraw the amount and deposit it into another account. We can then define four actions that a customer can make:

- Register: the customer opens a saving account
- Deposit(*amount*): the customer deposits *amount* to his account, where *amount* must be positive.
- Get: the customer reads the balance of his account.
- Withdraw(*amount*): the customer withdraw *amount* from his account. The amount must be positive and less than the balance.

From these actions, we can define the conflicting relation. We assume that as long as the client is registered, the client always sees the right balance (for simplicity). We can notice that no action commute with Register, except Register itself. Then, if a client deposits 10 and then deposits 20, it gives the same result as if he deposits 20 and then 10. It still holds if he trie to see his balance in-between. Finally, if a customer withdraws money and then deposits, the result is not the same as if he deposits first. Thus, we can define the following conflicting relation $\sim$ on the set $\mathcal{M} = \{Register, Deposit, Get, Withdraw\}$:

$$\begin{aligned} \sim = \ & \{(Register, Deposit), (Register, Get), (Register, Withdraw), \\ & (Deposit, Register), (Get, Register), (Withdraw, Register), \\ & (Deposit, Withdraw), (Withdraw, Deposit)\} \end{aligned} \tag{3}$$

It follows that only actions issued by the same customer conflict. The conflicting relation is implemented as a *ConflictingRelation* instance that defines an *is_related(cmd1, cmd2)* function according to (3). Note that the command contains an action and other metadata (e.g., a unique identifier, the identifier of the issuer, etc.)

The implementation in a replica is then straightforward. Each replica has an instance of a Banking system. When it executes a command, it performs the corresponding action on the banking system and sends the result to the client. The result might be a success (with eventually some data, e.g., the balance) or a failure. When a replica has to roll back a command, it simply executes the opposite one. For example, to roll back a Withdraw(10), it performs Deposit(10). We need to update data structures accordingly (e.g., a log file should remove the withdraw and not add the deposit).

4 Performance analysis

Now that we implement the generic broadcast protocol, it might be interesting to test its performance under given scenarios: under which condition is it worth it? We may first define some notions useful for the analysis.

- The transmission delay t is the time it takes for a message to travel from one peer to another.
- The consensus duration c is the total time taken by the Recovery consensus.
- The probability of conflict is the probability that a conflicting command appears in a set of commands. More formally, it is the probability that the protocol has to invoke a Recovery Consensus to process commands. For example, given that a client is registered, if he withdraws money every four deposits, the probability of conflict is $\frac{1}{5} = 20\%$.
- We introduce the concept of the slowdown factor sf that links the transmission delay and the consensus duration. Given that a set of non-conflicting commands has been processed (during the acknowledgment phase), it is the ratio between the time it takes to process a conflicting command and a non-conflicting one. If we assume that the transmission of information dominates the processing delay, we can link the transmission delay and the consensus duration as follows:

$$t = sf \cdot c \tag{4}$$

4.1 Expected latency

Consider that the network is distributed over the world (i.e., replicas are geographically far away). A good approximation of the expectation of transmission delay can be 300 milliseconds. According to [3], a good approximation about the expectation of the consensus duration for a small network is 5 seconds. We simulate the expectation of the time taken to process 100 commands (randomly issued by any client) on our network with 2 clients, 7 replicas (with one byzantine), and the communication delays given above as a function of the probability of conflict. Each command is chosen randomly and according to the probability of conflict to form a set of 100 commands. The commands are then executed by the network. We repeat this scenario ten times and use the mean of the sampled value. For technical reasons, we

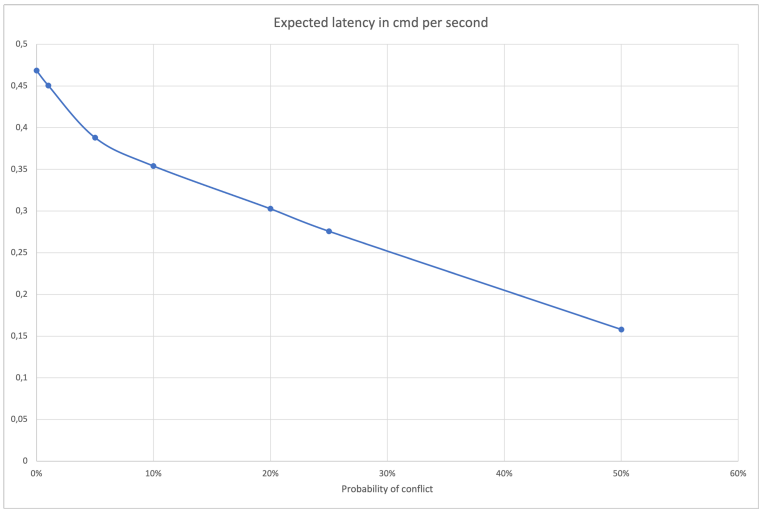


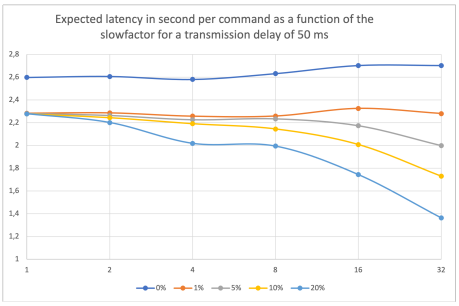
Figure 3: Expected latency in command executed per second as a function of the probability of conflict

simulate it for the following probability of conflict: 0% (best-case), 1%, 5%, 10%, 20%, 25% and 50%.

Figure 3. represents the results. One can observe that the expected latency decay exponentially for small probabilities and tends to decay linearly above 10%. The expected latency is divided by 2 (compared to the best-case, i.e. no conflict) for a probability of conflict that exceeds 25%. Depending on user requirements, we may assume that it is the maximal latency that a user can accept. Thus, the generic broadcast protocol tends to be useful only if less than one command over four conflicts. In particular, this might be a good approach for our saving accounts system. Customers tend to deposit more frequently than they withdraw money. Such a system could be useful to decentralize the truth (e.g., the truth is separated among log files in different replicas, and not contained by the bank databases).

4.2 Slowdown factor

The duration of the Recovery Consensus protocol may vary depending on the transmission delay, but also the network size and the optimizations of the algorithm (e.g., for a given use case and at the cost of memory consumption). We do not contribute to these relations, but it could be interesting to study the question. From now on, we will assume it is the case. Given that the transmission delay is well-known in a system, we can study whether this algorithm applies the same latency reduction for a given probability of



(a) transmission delay of 50 ms

(b) transmission delay of 200ms

Figure 4: Expected latency in command per second as a function of the slowdown factor for different probabilities of conflict

conflict and a consensus duration. In particular, we will compare it for transmission delays of 50 and 200 milliseconds, and for some realistic probabilities of conflict (we may assume that for more that 20% of conflict, no one would use this protocol). Figure 4.a shows the expected latency in command per second for different probabilities of conflict and a transmission delay of 50 ms. Figure 4.b shows the same data for a transmission delay of 200 ms.

One can notice that the scale factor for the transmission delay of 200ms gives the same consensus duration as the scale factor for the transmission delay of 50ms (i.e., $50 \cdot 1 = 200 \cdot 0.25$ and so on). The first thing to notice is that, for a scale factor of 1, the expected latency doesn't change for any probability of conflict. Thus, if the consensus durations is the same as the transmission delay, we don't get any benefit in the best-case, i.e., when no command conflict (the algorithm is not worth it anymore). On the other hand, for the same consensus duration and the same probability of conflict, the expected latency decays faster for the network with the fastest transmission delay (e.g., for a probability of conflict of 20%, the latency of the network with a transmission delay of 200ms is reduced by $\frac{0.08}{0.59} = 13.5\%$ and by $\frac{0.9}{2.3} = 40\%$ for the second network). Thus, it seems that a slower network tends to support a higher probability of conflict for the same latency reduction (i.e., the latency reduction is compared to the best-case latency).

Conclusion

The generic broadcast protocol proposed tends to be efficient for low probability of conflict. Implementing such a protocol is challenging as it requires many parametrizations. In particular, every command that a client can input must be known to define the conflicting relation. Our application to our

Banking systems appears to be an easy and simple example to understand the performance of this generic protocol. It would be interesting to compare the result obtained with an atomic broadcast protocol such as Honey Badger BFT or PBFT. However, note that our implementation may require some optimizations to be compared to other Byzantine Fault-Tolerant protocols.

References

[1] P. Raykov, N. Schiper, F. Pedone, Byzantine Fault-Tolerant with Commutative Commands, 2011.

[2] L. Lamport, R. Shostak, M. Pease, The Byzantine Generals Problem, 1982.

[3] A. Miller, Y. Xia, K. Croman, E. Shi, D. Song, The Honey Badger of BFT Protocols, 2016.

[4] The Rust Book, https://doc.rust-lang.org/book/

[5] Tokio library (Rust), https://tokio.rs

[6] M. Vidigueira, M. Monti, Talk library, https://github.com/Distributed-EPFL/talk, EPFL, 2021.

[7] Wikipedia, Byzantine fault, https://en.wikipedia.org/wiki/Byzantine_fault

[8] POA Network, How Honey Badger BFT Consensus Works.