6.824 Project
Practical Byzantine Consensus

Haoyue Wang & Boris Lipchin

May 11, 2014

1 Introduction

In class we have explored consensus tolerant of faults with a fail-stop failure model. This failure model allows for the possibility of machine failure, lossy networks and network partitions. Consensus in the presence of these failures can be effectively solved by Paxos protocols or derivatives. However, malicious attacks, software design errors and pathological random errors can cause elements of a distributed system to fail in a byzantine way. Multiple system elements controlled with malicious intent can even collude to break the system.

2 Approach

Byzantine fault tolerant consensus protocols are intended to address this problem, yet were long considered too expensive to be practical. In 1999, Miguel Castro and Barbara Liskov introduced the more efficient "Practical Byzantine Fault Tolerance" (PBFT) algorithm [1], which triggered a in BFT replication research.

In this project we implemented a PBFT algorithm in C/C++, and ran performance characterization in order to try and estimate the cost of Byzantine consensus over a loopback network. We used Unix domain sockets to transmit all of the information. The goal was to implement a consensus algorithm, while solving all of the detail-oriented memory issues that can be abstracted by higher order languages.

3 Design and Implementation

Similarly to Paxos 2-phases approach, our Byzantine consensus algorithm has three phases: pre-prepare, prepare, and accept. At the conclusion of the third phase, all replicas have agreed on a sequence number, corresponding client request, and the associated response. At the conclusion of the accept phase, all replicas transmit their response directly to the client. The client then, when it receives sufficient numbers of responses considers the operation to be completed. In order to maintain Byzantine fault-tolerance, a total of 3f + 1 replicas must be active initially, with at most f acting as Byzantine failed nodes.

A more formal description of the algorithm, we provide in the appendix below.

In order to drive our Byzantine consensus algorithm with an application, we implemented a key/value store, not unlikely the one we used for Paxos. Of course, the design of the entire system was substantially different from our lab approach. This was driven by the fact that the Byzantine consensus interaction between clients and replicas is quite different from Paxos, as well as the fact that our system is implemented in C/C++.

3.1 Network Authentication

A critical element where Byzantine consensus protocol differs significantly from less strict consensus algorithms is the use of cryptography. Byzantine consensus relies on majority of nodes on knowing what the majority of their peers are thinking, without any possibility of tampering. For example, if node b passes to node c a message from node a, node b must have a way of knowing that the message indeed came from node a, and was not tampered with or even created by b. Liskov's PBFT introduces the novel concept of authentication lists. Note: while not all Byzantine consensus protocols employ cryptographic hashes and encryption, it allows for more faults to occur per node on the system.

While it is not unusual for Byzantine consensus protocols to simply employ public-key encryption to guarantee safety of message passing, PBFT allows for a more efficient and weaker type of authentication. Every message in PBFT is signed with a list of n signatures for every n recipients that the message can reach. For example, a client sending a request to a set of replicas would sign its message with $3f + 1$ of such signatures. The algorithm requires that each such replica would share a secret key with each client, such that only replica i can validate the i'th signature in the message. While this requires every pair of nodes to share a common private key, this is a very solvable problem. Indeed, this approach saves significant processing time, as signing a message with a known private key is easier than doing public-key cryptography.

Our approach is to create an MD5 hash of the message to be signed, and simply XOR the result with the private key. This creates both a signature that is cryptographically secure, and relatively efficient to compute. We did not take the time to distribute private keys between nodes securely. Instead, we invented a hack where the private key between any two nodes of the system was a function of the node addresses and the node types. This allowed us to focus on the algorithm implementation rather than on semantics of securely distributing private keys over a network.

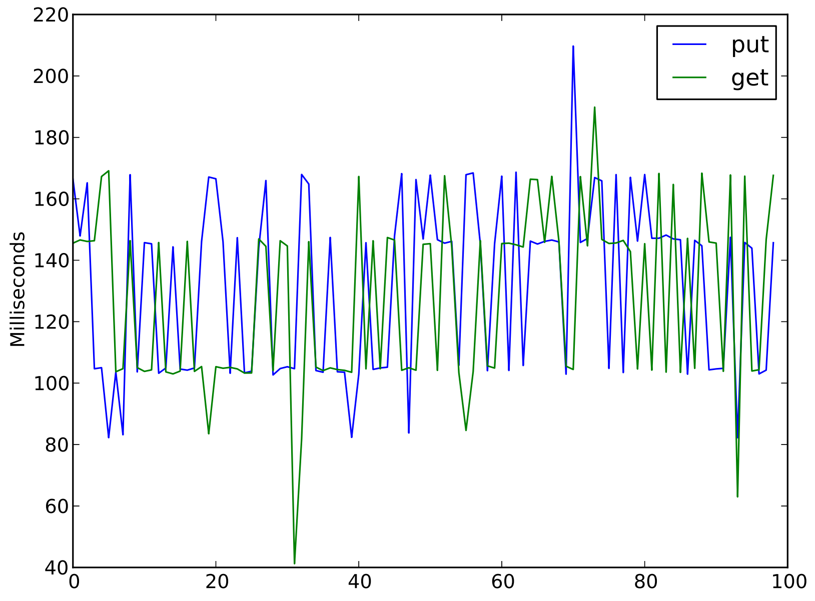
4 Results

We ran performance tests in order to characterize the performance of our Byzantine consensus implementation, and to understand how performance degraded as more nodes were added to the network. Since Unix domain sockets were used for all communication we can safely say that we eliminated most network-related delays from the system. This allows us to elucidate something akin to a theoretically speed limit for byzantine consensus, that is until concurrent file IO interactions through the current become so contentious that they begin to dominate the network latencies they are meant to replace. This, while it is an interesting exercise to achieve the fastest possible consensus, it is unlikely to scale well.

As the base line we conducted a series of PUT/GET calls on our Byzantine state machine. As we injected no faults for this test, this just serves as an speed upper bound for our algorithm. Performance analysis with Byzantine-style faults is performed further below.

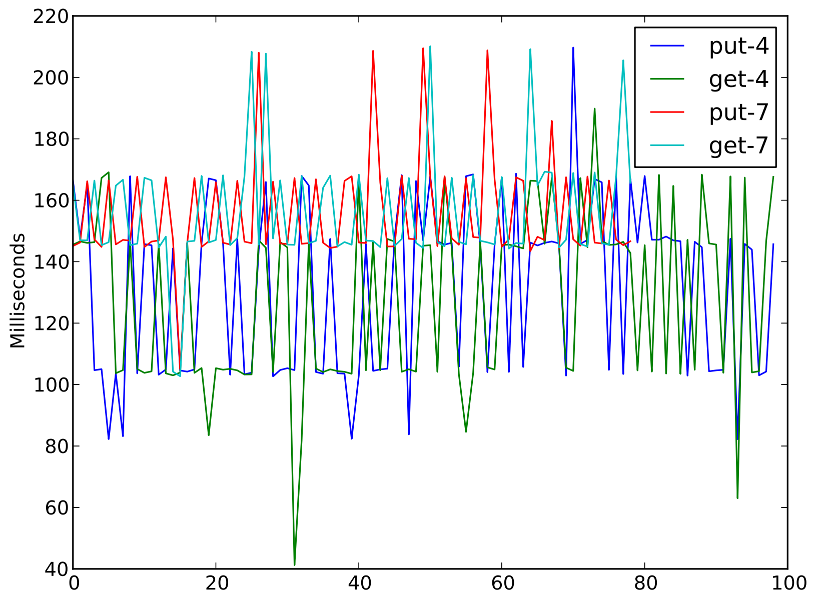
4.1 Performance Results

Figure 1: Nominal Put/Get Latency plot with 4 replicas



As we can see, when there are no network effects or failures imposed on the consensus, the performance is reasonably reliable.

Figure 2: Nominal Put/Get Latency plot with 7 replicas



Increasing the number of replicas to be robust to two faults rather than one did not affect the average latency and throughput, but did however increase the standard deviation of the message latency. This is to be expected as more resources of the system are being used by the consensus protocol.

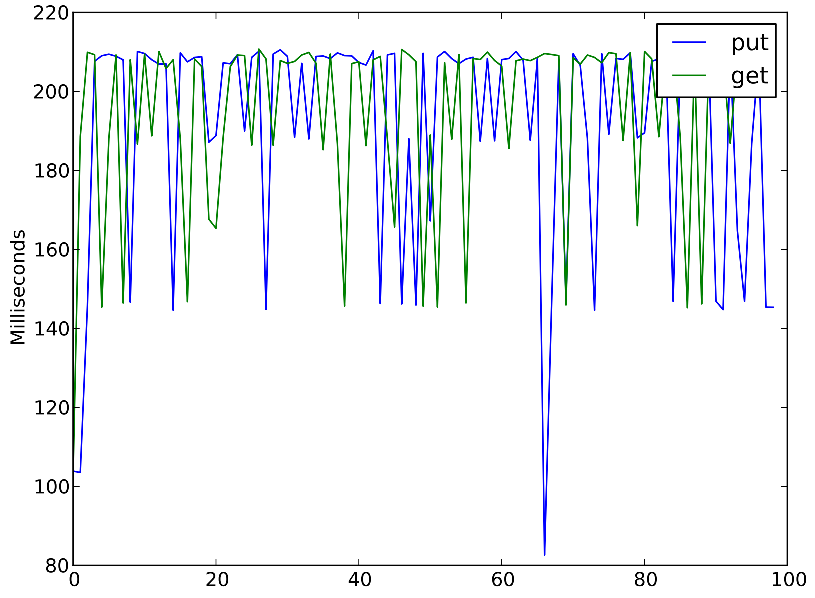
4.2 Byzantine Failure Testing

We also came up with a series of tests we believed to be pathological Byzantine. The assumption is that if the system can withstand the most pathological tests, it should be robust to a whole set of different attack vectors.

4.3 Silently drop all messages on a node

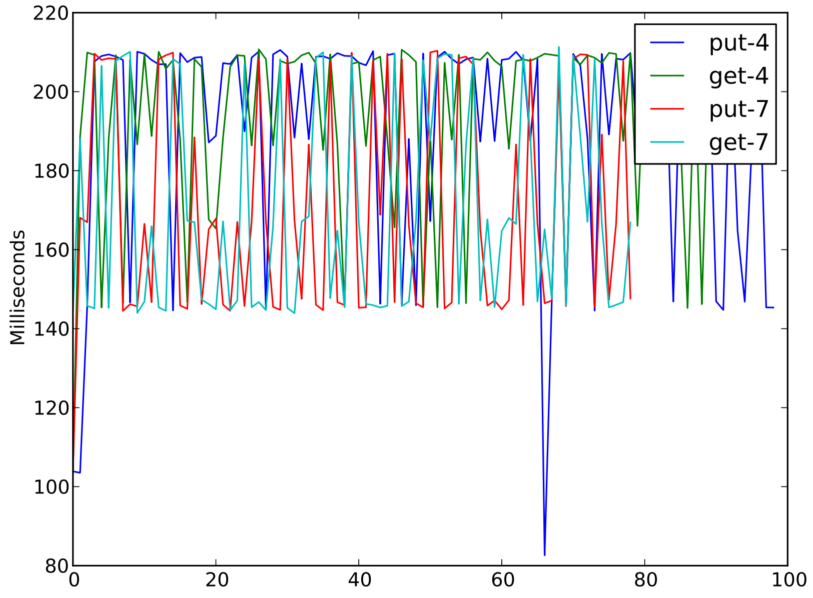
In this test we forced a node to completely drop all messages that it received. Since with 4 replicas, the remaining 3 non-faulty replicas become critical for the algorithm to reach consensus, this pegs the algorithm at the speed of the slowest replica. The performance change here is apparent.

Figure 3: Silent fail



In the graph below we suffered a similar failure, but with 7 total replicas, with 2 failed (rather than 4 and 1). It is interesting to note that despite the fact that network usage should have gone up almost exponentially with the number of nodes, latency remains the same. This means that the performance bottlenecks not at the loooback network, but most likely at the memory or processor level.

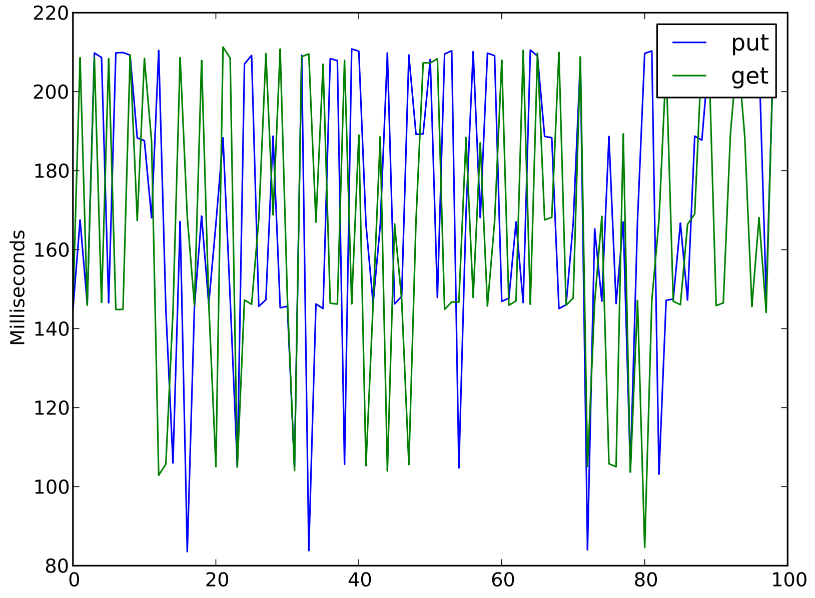
Figure 4: Silent fail for 2x nodes



4.4 Cheat the client in reply

In this test we forced a replica to return invalid reply messages back to the client, forcing a disagreement between replica nodes at the client. The client in this scenario will still choose the correct answer since the majority of replicas are non-faulty. However, the client is now required to wait for the slowest replica to respond, as it has now become critical for reaching consensus. The slowdown is comparable to the fail silent failure above, and demonstrates that Byzantine and non-Byzantine faults have similar impacts on the performance of the consensus algorithm.

Figure 5: Lies to Clients in Reply



5 Additional Testing

Other than injection of pathalogically failed nodes into the network, we used the Google Unit Test framework and generated a series of unit tests to test various subsystems, as well as several hard-to-mock up consensus problems. This ended up being hugely helpful to maintain regression, and find difficult bugs in our client/replica interactions.

A Algorithm Description Appendix

Symbol	Denotation
o	operation
t	timestamp
c	client
$\sigma_?$	signature by ?
v	the current view number
i	replica number
R	total number of replicas
p	primary $= v \ mod \ \lvert R \rvert$
r	the result of executing the requested operation
m	teh client's request message
$D(m)$	m's digest
$< m >_{\sigma_i}$	signing a digest of m and appending it to m
d	message digest
n	sequence number
C	a set of $2f + 1$ messages proving the correctness of checkpoint s
P	a set containing a set P_m for each request m that prepared at i with a sequence number higher than n
P_m	contains a valid pre-prepare message (without the corresponding client message) and $2f$ matching, valid prepare messages signed by different backups with the same view, sequence number, and digest of m
V	a set containing the valid view-change messages received by the primary plus the view change message for $v + 1$ the primary sent (or would have sent)
O	a set of pre-prepare messages (without the piggybacked request)

Actions taken by all parties are recorded below:

Client

1. Send $<\text{REQUEST}, o, t, c >_{\sigma_c}$ to the primary.
2. Accept the result r for request identified by c, t and v if the client receives $f + 1$ replies with valid signatures from different replicas, and with the same t, r, and v. Learn the current view number in this step (this is inferred).
3. Broadcast $<\text{REQUEST}, o, t, c >_{\sigma_c}$ to all replicas if does not receive replies soon enough.

Primary Replica

1. Primary replica shares most of the operations that other replicas have. If there is an exception, a blue note will be appended.
2. After receiving $<\text{REQUEST}, o, t, c >_{\sigma_c}$, i.e., m from the client or from a replica:

- multicast $<<$PRE-PREPARE$, v, n, d >_{\sigma_p}, m >$ to all other replicas and append the two mesages to its log

3. (This operation only applies to a replica that is the primary of a new view $v+1$.) If $2f$ valid $<$VIEW-CHANGE$, v+1, n, C, P, i >_{\sigma i}$ are received from other replicas:

   - Prepare O as:
     - The primary determines the sequence number min-s of the latest stable checkpoint in V and the highest sequence number max-s in a prepare message in V.
     - The primary creates a new pre-prepare message for view $v+1$ for each sequence number n between min-s and max-s. There are two cases:
       * There is at least one set in the P component of some view-change message in V with sequence number n. Create a new message $<$PRE-PREPARE$, v+1, n, d >_{\sigma_p}$, where d is the request digest in the pre-prepare message for sequence number n with the highest view number in V.
       * Otherwise, create a new message $<$PRE-PREPARE$, v+1, n, d^{null} >_{\sigma_p}$, where d^{null} is the digest of special null request, whose execution is no-op.
   - Multicast a $<$NEW-VIEW$, v+1, V, O >_{\sigma_P}$ message to all other replicas.
   - Append messages in O to its log.
   - If min-s is greater than the sequence number of its latest stable checkpointm, the primary insert the proof of stability for the checkpoint with sequence number min-s in its log, and discards infomation accordingly.
   - Enter view $v+1$ and start to accept messages for view $v+1$.

Non-Primary Replica

1. Data

   - a message log
   - view number v

2. After receiving a pre-prepare packet of two messages, i.e.,
   $<<$PRE-PREPARE$, v, n, d >_{\sigma_p}, m >$ from the primary (note this operation does not apply to the primary replica):

   - It accepts this message, multicasts $<$PREPARE$, v, n, d, i >_{\sigma i}$, and adds both messages to its log, provided (do nothing otherwise):
     - The signatures in the request and the pre-prepare message are correct and d is the digest for m.
     - It is in view v.
     - It has not accepted a pre-prepare message for view v and sequence number n containing a different digest.
     - $h <= n <= H$

3. After receiving a prepare message, i.e., $<$PREPARE$, v, n, d, i >_{\sigma i}$:

- The message will be added to the log if the signature is correct, the view number equals the replica's current view number, and the sequence number is between h and H
- Check if the predicate *prepared*(m, v, n, i) is ture, which is ture iff the log has:
  - the request m
  - a pre-prepare for m in view v with sequence number n
  - $2f$ prepares from different backups that match the pre-prepare (matching means they have the same view, sequence number, and digest)
- If the above predicate is ture, then multicasts $<$COMMIT$, v, n, D(m), i >_{\sigma i}$ to the other replicas.

4. After receiving a commit message, i.e., $<$COMMIT$, v, n, D(m), i >_{\sigma i}$:
   - The message will be added to the log if the signature is correct, the view number equals the replica's current view number, and the sequence number is between h and H
   - Check if the predicate *commited_local*(m, v, n, i) is ture, which is ture iff the log has:
     - $2f+1$ commits (possibly including its own) from different replicas that matches the pre-prepare for m; note a commit matches a pre-prepare if they have the same view, sequence number, and digest
   - If the above predicate is ture, execute the requested operation and send $<$REPLY$, v, t, c, i, r >_{\sigma_i}$ to the client. Replicas discard requests whose timestamp is lower than the timestamp in the last reply they sent to guarantee exactly-once semantics.

5. When a replica i produces a checkpoint, it multicasts a message $<$CHECKPOINT$, n, d, i >_{\sigma i}$ to the other replicas, where n is the sequence number of the latest request whose execution is reflected in the state and d is the digest of of the state.

6. If the replica has $2f+1$ (this is the number in [1], however, it is $f+1$ in [2]; [1] is newer than [2], so probably correct.) checkpoint messages in its log for sequence number n with the same digest d signed by different replicas (including possibly its own such message), discard all pre-prepare, prepare, and commit messages with sequence number less than or equal to n from its log. Also, discard all earlier checkpoints and checkpoint messages. Set h to be the sequence number of the last stable checkpoint, and set $H = h + K$, where k is big enough so that replicas do not shall waiting for a checkpoint to become stable.

7. After receiving $<$REQUEST$, o, t, c >_{\sigma_c}$ from the client:
   - If the request has already been processed, resend the reply; replicas remember the last reply messages they sent to each client.
   - Otherwise, relays the request to the primary.
     - Start a timer if the timer is not already running
     - Before the timer times out, if the replica executes the current request, it stops the timer. But if it is waiting for another pre-prepare from the primary, it restarts the timer immediately.
     - If the timer times out:

        * It stops accepting messages other than checkpoint, view-change, and new-view messages.
        * It multicasts <VIEW-CHANGE, $v+1, n, C, P, i >_{\sigma i}$

8. After receiving <NEW-VIEW, $v+1, V, O >_{\sigma_P}$:
    - Check if the message is signed properly.
    - Check if the view change it contains are valid for view $v+1$.
    - Check if the set O is correct by performing a compution similar to the one used by the primaryto create O.
    - If all checks pass, then:
        - Add the new information to its log as the new primary does.
        - Multicast a prepare for each message in O to all the other replicas, and Add these prepares to its log.
        - Enter view $v+1$